TF-IDF recap
- Represent the queries as vectors
- Represent the documents as vectors
- proximity = similarity of vectors

TF-IDF recap: document vectors

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>5.25</td>
<td>3.18</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
</tr>
<tr>
<td>Brutus</td>
<td>1.21</td>
<td>6.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>8.09</td>
<td>2.94</td>
<td>0</td>
<td>1.51</td>
<td>0.25</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1.54</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>2.65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1.51</td>
<td>0</td>
<td>1.0</td>
<td>0.12</td>
<td>0.25</td>
</tr>
<tr>
<td>winner</td>
<td>1.37</td>
<td>0</td>
<td>0.11</td>
<td>4.19</td>
<td>0.25</td>
</tr>
</tbody>
</table>

A document is represented by a vector of weights for each word.
TF-IDF recap: document vectors

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>5.25</td>
<td>3.18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.35</td>
</tr>
<tr>
<td>Brutus</td>
<td>1.21</td>
<td>0.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>8.99</td>
<td>2.54</td>
<td>0</td>
<td>1.51</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1.54</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>2.85</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1.51</td>
<td>0</td>
<td>1.9</td>
<td>0.12</td>
<td>5.25</td>
<td>0.38</td>
</tr>
<tr>
<td>waror</td>
<td>1.37</td>
<td>0</td>
<td>0.11</td>
<td>4.15</td>
<td>0.25</td>
<td>1.95</td>
</tr>
</tbody>
</table>

One option for this weighting is TF-IDF:

\[ w_{i,d} = \text{tf}_{i,d} \times \log(N/\text{df}_i) \]

TF-IDF recap: similarity

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

Dot product

Unit vectors

\[ \cos(q,d) = \frac{\sum_{i=1}^{F} q_i d_i}{\sqrt{\sum_{i=1}^{F} q_i^2} \sqrt{\sum_{i=1}^{F} d_i^2}} \]

\[ \cos(q,d) \] 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

weights

\[ \text{doc} \]

How do we do this?

\[ \text{query} \]

\[ \cos(\mathbf{q}, \mathbf{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \cdot \sqrt{\sum_{i=1}^{n} d_i^2}} \]

Calculating cosine tf-idf from index

weights

\[ \text{index} \]

What should we store in the index?

\[ \text{w}_1 \]

How do we construct the index?

\[ \text{w}_2 \]

How do we calculate the document ranking?

\[ \text{w}_3 \]

\[ \ldots \]

\[ w_{\text{tf-idf}} = \text{tf}_{i,j} \times \log \left( \frac{N}{d_{f_{i,j}}} \right) \]

\[ \cos(\mathbf{q}, \mathbf{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \cdot \sqrt{\sum_{i=1}^{n} d_i^2}} \]
Index construction: collect docIDs

Doc 1
- Caesar was ambitious
- Brutus hath told you

Doc 2
- So let it be with Caesar. The noble Brutus hath told you
- Caesar was ambitious

Index construction: sort dictionary

sort based on terms

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?

Do we have all the information we need?
An aside: speed matters!

- 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?

- 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
        score[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
        score[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] += \( w_{t,q} \times w_{t,d} \) more expensive

return top k components of scores sort?

Why is this more expensive than boolean?

Key challenges for speedup

boolean

Intersection

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] += \( w_{t,q} \times w_{t,d} \) more expensive

return top k components of scores sort?

Why is this more expensive than boolean?
Key challenges for speedup

- Interrogation
- rank

- Intersection

- 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
  - 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

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 $N = 10^6$, $K = 100$, 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

Documents
Score documents
Pick top K

Approximate approach

Documents
Select A candidates
K < A << N
Score documents in A
Pick top K in A

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
Approximate
Exact vs. approximate

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

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

Documents with more than one 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?

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>13</th>
<th>21</th>
<th>34</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brutus</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Calpurnia</td>
<td>1</td>
<td>3</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scores only computed for 8, 16 and 32.

Multiple query terms

How many documents have we “pruned” or ignored?

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>13</th>
<th>21</th>
<th>34</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brutus</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>64</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Calpurnia</td>
<td>1</td>
<td>3</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 **tiers**

### Tiered indexes

Break postings up into a hierarchy of lists
- Most important
- ...
- Least important

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**

<table>
<thead>
<tr>
<th>Tier 1</th>
<th>Tier 2</th>
<th>Tier 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>auto</td>
<td>auto</td>
</tr>
<tr>
<td></td>
<td>best</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>car</td>
<td>car</td>
</tr>
<tr>
<td></td>
<td>insurance</td>
<td>insurance</td>
</tr>
<tr>
<td>Doc2</td>
<td>Doc1</td>
<td>Doc1</td>
</tr>
<tr>
<td>Doc1</td>
<td>Doc2</td>
<td>Doc3</td>
</tr>
<tr>
<td>Doc3</td>
<td>Doc2</td>
<td>Doc2</td>
</tr>
</tbody>
</table>

**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 **more than one** 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?

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

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

\[
\begin{align*}
\text{Antony} & \rightarrow 1 \rightarrow 2 \\
\text{Brutus} & \rightarrow 3 \rightarrow 1 \rightarrow 2 \\
\text{Caesar} & \rightarrow 3 \rightarrow 2 \\
\end{align*}
\]

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

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

\[
\begin{align*}
\text{Antony} & \rightarrow 1 \rightarrow 2 \\
\text{Brutus} & \rightarrow 3 \rightarrow 1 \rightarrow 2 \\
\text{Caesar} & \rightarrow 3 \rightarrow 2 \\
\end{align*}
\]

\(g(1) = 0.5, \ g(2) = .25, \ g(3) = 1\)
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 $g(d) + \text{tf-idf}_d$
- 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?