Homework 4 – things that seemed to work

• Bidirectional LSTM instead of unidirectional
• Change LSTM activation to sigmoid
• Larger embedding size, smaller hidden layer size
• Try different optimizers
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).

• The most common task is **web search**, but there are others:
  • Email search
  • Searching from laptop
  • Corporate knowledge bases
  • Legal information retrieval
Basic assumptions of information retrieval

• **Collection**: a set of documents
  • Can be a fixed set or dynamically changing
  • Assume it is static for now

• **Goal**: Retrieve documents with information that is relevant to the user’s information need and helps the user to complete a task
Ad-hoc information retrieval task

User task

Info need

I want to cook a good Bolognese sauce

Good Bolognese sauce recipes

best bolognese recipe

Query refinement

Search engine

Result

Collection

Query

I want to cook a good Bolognese sauce

Good Bolognese sauce recipes

best bolognese recipe
The Plan

• Boolean information retrieval
• Ranked information retrieval
• Evaluating IR systems

• Slides mostly based on:
Boolean information retrieval
Information retrieval from unstructured data

• Which Shakespeare’s plays in Shakespeare’s Collected Works contain the words Brutus and Caesar but not Calpurnia?

• Read through all the text linearly using for instance `grep` to find documents containing Brutus and Caesar and then strip out documents that contain Calpurnia

• Not a general solution
  • Slow for large document collection
  • Want to use more flexible operators, e.g. Romans NEAR countrymen
  • Want to receive results ranked by relevance

• The general solution is to index each document in advance
Term-document matrix

- Record all words in all documents in a term-document matrix

<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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean information retrieval

• Which Shakespeare’s plays in Shakespeare’s Collected Works contain the words Brutus and Caesar but not Calpurnia?

Brutus: 110100 AND
Caesar: 110111 AND
NOT Calpurnia: 101111 = 100100

Answer:
Antony and Cleopatra
Hamlet
Large collections

• Suppose we have 1 million documents
• Suppose each document is ca 1000 words long
• Assuming 6 bytes per word on average, the size of this document collection is ca 6GB
• Assuming 500 000 different words/terms, what is the size of the corresponding boolean term-document matrix?
Better representation?

• Assuming every number takes 1 bit the size of the matrix is ca 60GB

• However, the matrix is sparse, containing only at most 1 billion \((10^9)\) ones, out of \(5 \times 10^{11}\) numbers --> at least 99.8% entries are 0

• Thus, record only things that occur (1’s) and not things that do not occur (0’s)
**Inverted index**

- **Inverted index** is the key data structure in any information retrieval system.
Inverted index

• Vocabulary keys are sorted
• Document id’s are sorted in postings
• Vocabulary is commonly stored in memory
• Postings are stored on disc
• There is a pointer from each vocabulary item to the corresponding posting list
Building an inverted index

1. Collect the documents to be indexed
   - Friends, Romans, countrymen.
   - So let it be with Caesar

2. Tokenise each document into a list of tokens
   - Friends
   - Romans
   - countrymen
   - So
   - let
   - it
   - be
   - with
   - Caesar

3. Linguistic preprocessing, normalisation
   - friend
   - roman
   - countryman
   - so
   - let
   - it
   - be
   - with
   - caesar

4. Construct the inverted index
Text preprocessing, normalization

1. Normalization
   • Maps different forms of the same word/term (e.g. U.S.A. is the same as USA)

2. Stemming
   • Cluster the inflectional forms of the same lemma, e.g. reads, read, reading
   • Cluster the words with the same root, e.g. authorize, authorization

3. Stop words
   • Maybe want to remove stop words (but maybe not)
Constructing the inverted index

• Input to the indexing system is a list of normalised (term, docID) pairs

<table>
<thead>
<tr>
<th>term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i’</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>

Doc1
I did enact Julius Caesar: I was killed i’ the Capitol; Brutus killed me.

Doc2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:
Constructing the inverted index

- Sort the list by word and then by docID

<table>
<thead>
<tr>
<th>term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i’</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
Constructing the inverted index

- Construct dictionary and postings
Processing a Boolean query with inverted index

• Consider processing the query: **Brutus AND Caesar**
• Locate **Brutus** from the dictionary and retrieve its posting list
• Locate **Caesar** from the dictionary and retrieve its posting list

![Diagram showing postings for Brutus and Caesar]

• “Merge” the two postings (intersect the document sets)
Merge the postings

• Process two posting lists simultaneously from left to right to find matching document ids.

Brutus

Caesar
Merge the postings

- Process two posting lists simultaneously from left to right to find matching document ids.

- The complexity is $O(x+y)$, where $x$ and $y$ are the lengths of both lists correspondingly.

- The sorted order of postings is crucial!
The “merge” algorithm

\[
\text{INTERSECT}(p_1, p_2)
\]

1. \( \text{answer} \leftarrow \langle \rangle \)

2. \textbf{while } p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL}

3. \textbf{do if } \text{docID}(p_1) = \text{docID}(p_2)

4. \hspace{1em} \textbf{then } \textbf{ADD}(\text{answer}, \text{docID}(p_1))

5. \hspace{2em} p_1 \leftarrow \text{next}(p_1)

6. \hspace{2em} p_2 \leftarrow \text{next}(p_2)

7. \hspace{1em} \textbf{else if } \text{docID}(p_1) < \text{docID}(p_2)

8. \hspace{2em} \textbf{then } p_1 \leftarrow \text{next}(p_1)

9. \hspace{2em} \textbf{else } p_2 \leftarrow \text{next}(p_2)

10. \textbf{return } \text{answer}
Phrase queries

• Often people want to search for an exact phrase such as The Who
• In this case the sentence The man who knew infinity should not be a match
• In web search engines exact phrases can be enclosed in double quotes: “The Who”
• For performing phrase search queries storing term:doc entries is not enough
Bigram indexes

• Index every consecutive term pair in the text as a phrase
• For example the text “Friends, Romans, countrymen” would generate the bigrams:
  • friends romans
  • romans countrymen
• Add all these bigrams into dictionary
• Two-word phrase processing is now immediate
Longer phrase queries

• Longer queries can be processed by breaking them into bigrams

• university of tartu can be broken into the Boolean query on bigrams:
  • university of AND of tartu

• However, without explicitly looking into the document we cannot verify that the phrase really occurs in the document
Issues with bigram indexes

• False positives

• Index gets large due to bigram dictionary
  • Big even with bigrams, infeasible for longer phrases

• Bigram indexes are not the standard solution but can be part of a compound strategy
Positional indexes

• In the postings, store for each term the positions where it appears

<term, document frequency:
   doc1: position1, position2, ...;
   doc2: position2, position2, ...;
   ...
   ...
>
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 these documents could contain the phrase “to be or not to be”?

• For phrase queries, use merge algorithm recursively at the document level
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 of “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, ...

• The same general method can be used for proximity searches
Positional index size

• A positional index expands postings storage *substantially*
  • Even though indexes can be compressed
• Need an entry for each word occurrence, not just once per document
• Index size depends on average document size
  • Average web page has <1000 terms
  • Books, articles, legal documents can have much more words
• Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Rules of thumb

• A positional index is 2-4 as large as non-positional index

• Positional index size is 35-50% of the original text’s volume

• These numbers can differ when the language is other than English
Combine bigram and positional indexes

- Bigram and positional indexes can be combined
- For particular frequently searched phrases such as “Michael Jackson” or “Elon Musk” it is inefficient to keep merging positional postings lists

1. Can choose to store some phrases in a bigram index during preprocessing
2. Can choose to add frequently searched phrases dynamically to a bigram index
Boolean information retrieval

• Documents either match or they don’t
• Good for expert users with precise understanding of their needs and the collection
• It can be also good if the IR system is part of a larger application that can make use of 1000s of results
• Not good for majority of users
  • Most users are not capable of writing Boolean queries
  • Most users don’t want to get back 1000s of results
Problems with Boolean IR

• Boolean queries often result in too few or too many results

• Query1: “standard user dlink 650” --> 200000 hits
• Query2: “standard user dlink 650 not 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 information retrieval
Ranked retrieval models

- In **ranked retrieval models** the system returns an **ordering** over the (top) documents in the collection with respect to a query.
- Supports **free text queries**: instead of **boolean query** the query is now just a one or more words in a human language.
- Large search results are not an issue:
  - The size of the result set is not a problem because the user is shown only top $k$ results.
- This all assumes that the ranking algorithm works reasonably well.
Scoring as the basis of ranked retrieval

- The system should return the query results in an order that is useful to the searcher
- Have to rank the retrieved documents with respect to the query
- Assign a score (for instance between $[0, 1]$) to each document
- This score measures how well the document and the query match
Query-document matching scores

• How to assign a score to a query/document pair?

• Suppose that the query only contains one term
• If the query term does not occur in the document then the score should be 0
• The more frequent the query term occurs in the document the higher should be the score
Scoring boolean vectors

- In boolean term-document matrix each document is a binary vector

<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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Scoring with Jaccard coefficient

• Jaccard coefficient is a common measure of overlap between two sets $A$ and $B$

  • $\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$
  • $\text{Jaccard}(A, A) = 1$
  • $\text{Jaccard}(A, B) = 0$ if $|A \cap B| = 0$

• $A$ and $B$ don’t have to be the same size
• Jaccard coefficient always assigns a number between 0 and 1
Jaccard coefficient: example

• What is the query-document match score using the Jaccard coefficient?

• **Query**: ides of march
  • **Document 1**: caesar died in march
  • **Document 2**: the long march

• Jaccard(Query, Document1) =
• Jaccard(Query, Document2) =
Jaccard coefficient: example

• What is the query-document match score using the Jaccard coefficient?

• **Query:** *ides of march*
  • **Document 1:** *caesar died in march*
  • **Document 2:** *the long march*

• Jaccard(Query, Document1) = 1/6
• Jaccard(Query, Document2) = 1/5
Issues with Jaccard scoring

• It doesn’t consider the frequency of the terms (how many times a term occurs in the document)
• Rare terms in the document are more informative than the frequent terms
• The score is affected by the document length
Term-document count matrix

- Record the number of occurrences of each term in each document

<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>157</td>
<td>73</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brutus</td>
<td>4</td>
<td>157</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>232</td>
<td>227</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>57</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Bag-of-words model

• Vector representation doesn’t consider the word order in the documents
• John is quicker than Mary has the same vector as Mary is quicker than John
• Need to add also the positional information into the model
Term frequency

• The term frequency $tf_{t,d}$ is the number of times the term $t$ occurs in document $d$

• However, the raw $tf$ is perhaps not what we want
  • A document with 10 occurrences of the same word is more relevant than a document with 1 occurrence of the term
  • But not 10 times more relevant

• We want a $tf$ measure that increases monotonically when the raw term frequency increases but that does not increase linearly
Log-frequency weighting

• The log-frequency weight of term $t$ in $d$ is:
  • $w_{t,d} = 0$, if $tf_{t,d} = 0$
  • $w_{t,d} = 1 + \log tf_{t,d}$, if $tf_{t,d} > 0$

• Computing the score for a query-document pair:
  • Sum over terms $t$ in both query and document

\[
\text{Score} = \sum_{t \in q \cap d} (1 + \log tf_{t,d})
\]

• If none of the query terms is present in the document then score is 0
Document frequency

- **Rare words** are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the document collection (like *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- Thus, we want a higher weight for rare terms such as arachnocentric
Document frequency

• Frequent terms are less informative than rare terms
• Consider a query term that is frequent in the collection (e.g. high, increase, line)
• A document containing such a term is more likely to be relevant than a document that doesn’t
• But it’s not a sure indicator for relevance
• Thus, for frequent terms we want positive weights for words like high, increase and line
• But they should be lower than for rare terms
• Document frequency is used to capture that
Inverse document frequency

• $df_t$ is the document frequency of term $t$: the number of documents that contain $t$
  • $df_t$ is an inverse measure of the informativeness of $t$
  • $df_t \leq N$, the total number of documents

• The inverse document frequency is defined as:

$$idf_t = \log \frac{N}{df_t}$$

• The log is used to dampen the effect of idf
### Idf example, suppose $N = 1,000,000$

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>sunday</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>fly</td>
<td>10,000</td>
<td></td>
</tr>
<tr>
<td>under</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>1,000,000</td>
<td></td>
</tr>
</tbody>
</table>

$$idf_t = \log \frac{N}{df_t}$$
Idf example, suppose $N = 1000000$

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>sunday</td>
<td>1000</td>
<td>3</td>
</tr>
<tr>
<td>fly</td>
<td>10000</td>
<td>2</td>
</tr>
<tr>
<td>under</td>
<td>100000</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1000000</td>
<td>0</td>
</tr>
</tbody>
</table>

• 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 example, for the query “capricious person”, idf weighting makes occurrences of capricious more important in the final document ranking than the occurrences of person
Tf-idf weighting

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

\[ w_{t,d} = (1 + \log tf_{t,d}) \times \log \frac{N}{df_t} \]

• It is the best known weighting scheme in information retrieval
• Tf-idf increases with the number of term occurrences in a document
• Tf-idf increases with the rarity of the term in the collection
Final ranking of documents for a query

\[ \text{score}(q, d) = \sum_{t \in q \cup d} \text{tf-idf}_{t,d} \]
Weight matrix

- Each document is now represented by a real-valued vector

<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>6.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>8.59</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>0.88</td>
</tr>
<tr>
<td>worser</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>
Documents as vectors

• Now we have a $|V|$-dimensional vector space
• Terms are the axes of that space
• Documents are points or vectors in this space
• The space is very high-dimensional: it can have tens of millions of dimensions when applied to a web search engine
• The vectors are very sparse: most entries are zero
Queries as vectors

• **Key idea 1**: Represent queries also as vectors in the same term-space
• **Key idea 2**: Rank documents according to their proximity to the query in that space
• Proximity = similarity of vectors $\cong$ inverse of distance
• **Recall**: we do this because we want to get away from the *you’re either in or out* Boolean model
• Instead we want to rank more relevant documents higher than less relevant documents
Formalizing vector space proximity

- Distance between two points = distance between the end points of two vectors
- **Euclidean distance**?

- Euclidean distance is a bad idea because Euclidean distance is **large** for vectors of different lengths
Why Euclidean distance is a bad idea?

The Euclidean distance between vectors $q$ and $d_2$ is large even though the distribution of terms in the query $q$ and the distribution of terms in the document $d_2$ are very similar.
Use angle instead of distance

• If two document vectors point to the same direction then they are semantically similar ...

• ... even when the Euclidean distance between them is large (because one document is much longer than the other, for instance)

• Thus: rank documents according to the angle between the query and the document
From angles to cosines

• The following two notions are equivalent
  • Rank documents in increasing order of the angle between the query and the document
  • Rank documents in decreasing order of cosine between query and document

• Cosine is monotonically decreasing function for the interval [0°-180°]

• Very efficient to compute between two vectors
Length normalization

• A vector can be length-normalized by dividing its components by the vector length
• Vector length – L2 norm: \[ \| \mathbf{x} \|_2 = \sqrt{\sum_i x_i^2} \]
• Dividing a vector by its length makes it a unit-length vector
• After length-normalization long and short documents now have comparable weights
Cosine(query, document)

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

• \(q_i\) is the tf-idf weight of the term \(i\) in the query
• \(d_i\) is the tf-idf weight of the term \(i\) in the document
• For length-normalized vectors, cosine similarity is simply the dot product (or scalar product)

\[
\cos(q, d) = q \cdot d = \sum_{i=1}^{|V|} q_i d_i
\]
Cosine similarity illustrated
Information retrieval with ranking model

\[
\text{CosineScore}(q)
\]

1. \text{float} \ Scores[N] = 0
2. \text{float} \ Length[N]
3. \text{for each} \ query \ term \ t
4. \text{do} \ calculate \ \mathit{w}_{t,q} \ \text{and fetch postings list for} \ t
5. \hspace{1em} \text{for each} \ \mathit{pair}(d, \mathit{tf}_{t,d}) \ \text{in postings list}
6. \hspace{2em} \text{do} \ \text{Scores}[d] + = \mathit{w}_{t,d} \times \mathit{w}_{t,q}
7. \text{Read the array} \ Length
8. \text{for each} \ d
9. \text{do} \ \text{Scores}[d] = \text{Scores}[d]/\text{Length}[d]
10. \text{return} \ \text{Top} \ K \ \text{components of} \ \text{Scores[]}
## Tf-idf variants

<table>
<thead>
<tr>
<th>Term frequency</th>
<th>Document frequency</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>n (natural)</td>
<td>t (idf)</td>
<td>n (none)</td>
</tr>
<tr>
<td>l (logarithm)</td>
<td>p (prob idf)</td>
<td>c (cosine)</td>
</tr>
<tr>
<td>a (augmented)</td>
<td></td>
<td>u (pivoted unique)</td>
</tr>
<tr>
<td>b (boolean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L (log ave)</td>
<td></td>
<td>b (byte size)</td>
</tr>
</tbody>
</table>

| TF-IDF Variants                          | \(\text{TF-IDF Weighting with Cosine Normalization} \)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n (natural)</td>
<td>(\text{TF-IDF} )</td>
</tr>
<tr>
<td>l (logarithm)</td>
<td>(1 + \log(\text{TF-IDF}))</td>
</tr>
<tr>
<td>a (augmented)</td>
<td>(0.5 + \frac{0.5 \times \text{TF-IDF}}{\max_t(\text{TF-IDF})})</td>
</tr>
<tr>
<td>b (boolean)</td>
<td>(\begin{cases} 1 &amp; \text{if } \text{TF-IDF} &gt; 0 \ 0 &amp; \text{otherwise} \end{cases})</td>
</tr>
<tr>
<td>L (log ave)</td>
<td>(\frac{1 + \log(\text{TF-IDF})}{1 + \log(\text{ave}_{t,d}(\text{TF-IDF}))})</td>
</tr>
</tbody>
</table>

- **Itc** – tf-idf weighting with cosine normalization
Weighting in queries and documents

• The weighting can differ in documents and queries
• **SMART notation**: denotes the combination in use in a search engine with a notation **ddd.qqq**, using the acronyms from the previous table
• A very standard weighting scheme is **Inc.ltn**
• **Document**: logarithmic tf, no idf, cosine normalization
• **Query**: logarithmic tf, idf, no normalization
Summary – ranked retrieval models

• Represent the query as a weighted tf-idf vector
• Represent each document as a weighted tf-idf vector
• Compute the cosine similarity between the query and each document vector
• Rank documents with respect to query by score
• Return the top K documents to the user
Evaluating IR systems
Evaluating an IR system

• An information need is translated to a query

• Relevance is assessed relative to the information need not the query

• E.g. information need: *I’m looking for information on whether drinking red wine more effective at reducing the risk of heart attacks than white whine*

• Query: **wine red white heart attack effective**
Evaluating ranked results

• Evaluation of a result set
  • If we have:
    • A benchmark document collection
    • A benchmark set of queries
    • Annotated judgements which documents are relevant to which queries
  • Then we can use Precision/Recall/F-score on that set

• Evaluation of ranked results:
  • The system can return any number of results
  • By evaluating the increasing sets of top returned results we can construct precision-recall curves
Precision/Recall

• Assume there are 10 relevant documents in the collection

<table>
<thead>
<tr>
<th>No</th>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Precision/Recall

- Assume there are 10 relevant documents in the collection

<table>
<thead>
<tr>
<th>No</th>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>0.33</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td>7</td>
<td>R</td>
<td>0.57</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>N</td>
<td>0.5</td>
<td>0.4</td>
</tr>
<tr>
<td>9</td>
<td>N</td>
<td>0.44</td>
<td>0.4</td>
</tr>
<tr>
<td>10</td>
<td>N</td>
<td>0.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Mean Average Precision (MAP)

- AP: Average Precision of the top k retrieved documents computed at each time a relevant document is retrieved
- MAP: average AP’s for a set of queries
- Macro-averaging: each query counts equally

<table>
<thead>
<tr>
<th>Label</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1.0</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>0.5</td>
</tr>
<tr>
<td>R</td>
<td>0.6</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>0.57</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.67</td>
</tr>
</tbody>
</table>
IR book

• Introduction to Information Retrieval
  • By Manning, Raghavan and Schütze

• Available online: https://nlp.stanford.edu/IR-book/