4. Term Weighting and Vector Space Model

Prof. Dr. Goran Glavaš

Center for AI and Data Science (CAIDAS)
Fakultät für Mathematik und Informatik
Universität Würzburg

After this lecture, you'll...

- Be familiar with your first ranked retrieval model (VSM)
- Understand the TF-IDF term weighting scheme
- Know how to rank documents according to cosine similarity
- Know about some methods for speeding up VSM's ranking
- Be familiar with the multi-criteria ranking

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Recap of the previous lecture

- Data structures for inverted index
  - **Q:** What are the different data structures we may use for indexing?
  - **Q:** How do we build index with a hash table (pros and cons)?
  - **Q:** How do we build index with a balanced tree (pros and cons)?
- Tolerant retrieval: wild-card queries
  - **Q:** What are the different options for handling wild-card queries?
  - **Q:** What is a permuterm index and how do we use it for wild-card queries?
  - **Q:** How to use character indexes to support wild-card queries?
- Tolerant retrieval: error correction
  - **Q:** How to correct the spelling by observing the terms in isolation?
  - **Q:** How do we use the edit distance to fix for misspellings?
  - **Q:** What are the different options for spelling correction in context?

Recap of the previous lecture(s)

- **Inverted index** is a data structure for computationally efficient retrieval
- We've examined different variants of the inverted index for different queries
  - Regular inverted index for simple Boolean queries
  - Positional index for phrase and proximity queries
  - Permuterm index for tolerant retrieval
- Boolean retrieval has a major drawback
  - The results are not ranked
  - Without ranking: either too few or too many results
- Document d_j is represented by term vector $[w_{1j}, w_{2j}, \ldots, w_{tj}]$ where t is the number of index terms
  - Let g be the function that computes the weights, i.e., $w_{ij} = g(k_i, d_j)$
  - Different choices for the weight-computation function g and the ranking function r define different IR models
- **Today, we examine the first model for ranked retrieval – vector space model (VSM)**
  - **We examine what g and r are for VSM**

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Beyond Boolean retrieval

- So far, all our queries were some variant of Boolean (simple, phrase, positional)
  - Document either match or not
- Suitable for expert users with precise understanding of both
  - Their information needs
  - The document collection against which they spawn queries
- Also suitable for applications: easily consume 1000s of results
- Not suitable for most human users
  - Most users find it difficult (unnatural) to write Boolean queries
  - Most users cannot go through thousands of results the Boolean retrieval engine returns on large collections (e.g., web)

Beyond Boolean retrieval

- Boolean queries often yield either too few (even 0) or too many (1000s) results
  - Q1: „standard user dlink 650”
    - 200K hits
  - Q2: „standard user dlink 650 no card found”
    - 0 hits

- It takes a lot of skill, experience, and sometimes time to design a query that produces a manageable number of hits
  - AND operator often drastically reduces the number of hits
  - OR operator often drastically increases the number of hits
  - Hard to find the balance

- Solution: rank the documents and return the top N ranked hits
  - User directly chooses N, i.e., how many hits to process

Ranked retrieval

- IR models for ranked retrieval
  - Produce the ordering over the documents in the collection
- Selection of top-ranked documents
  - May be done by the IR system
    - Cut the documents below rank N (i.e., top N)
    - Cut documents below some treshold score value
  - May be left to the user
    - Entire ranking is returned (e.g., with paging)
- Free text search
  - No query language with operators and expressions
  - Query is simply one or more words in natural language
- Two separate design-decisions, but often go together
  - Free text search & ranked retrieval

Ranked retrieval

- Assumption: The ranking of the documents is based on the relevance
  - The ranking/scoring function (*r*) captures the extent of relevance of the document for the query
- All IR models that we will cover from now onwards are ranked retrieval models
  - They differ in the scoring function *r* they use

- Common-sense assumptions
  - Let's start from a single-term query q_t
  - If the term does not occur in the document d – $r(q_t, d) = 0$
  - The more frequent the query term in the document, the higher the score should be
    - $r(q, d) \propto f_{t,d}$

Ranked retrieval: naive approach

- First idea: use Jaccard coefficient – a measure of overlap of two sets A and B:

$$\text{Jaccard}(A, B) = |A \cap B| \,/\, |A \cup B|$$

$$\text{Jaccard}(A, A) = 1$$

$$\text{Jaccard}(A, B) = 0 \text{ iff } A \cap B = \emptyset$$

- The Jaccard index is always between 0 and 1
- Sets A and B don’t have to be of the same size
- Shortcomings of using Jaccard coefficient as a scoring function
  1. Term frequency in each of the documents is not taken into account
  2. The overall frequency of the term in the collection (or language in general) is not accounted for – rare terms are more informative
  3. There are more sophisticated ways to normalize for the document length

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Term frequency

- Term frequency *tf(t,d)* is a measure that denotes how frequently the term *t* appears in the document *d*
- **Q:** Shall we use the raw frequency (i.e., raw number of occurrences of *t* in *d*) as a measure of term frequency?
  - A document d_1 with 10 occurrences of a query term *t* is probably more relevant than a document d_2 with 1 occurrence. But is it 10 times more relevant?
  - A document d_1 contains 100.000 tokens and 4 occurrences of term *t* whereas the document d_2 contains 500 tokens and 3 occurrences of term *t*. Which document is more relevant?
- Relevance does not increase linearly with term frequency
- Raw term frequency does not account for document length

Term frequency

- Let's fix for the previous two observations

1. Relevance does not increase linearly with term frequency
   - Let's take the logarithm of the raw frequency

     $tf(t,d) = 1 + \log_{10}(f_{t,d})$, if $f_{t,d} > 0$, otherwise 0
2. Raw term frequency does not account for document length
   - Let's normalize with the frequency of the most frequent term in the document

     $tf(t,d) = f_{t,d} / \max\{f_{t',d} : t' \in d\}$

- Combining the two:

  $tf(t,d) = (1 + \log_{10}(f_{t,d})) \;/\; (1 + \log_{10}(\max\{f_{t',d} : t' \in d\}))$
  - if $f_{t,d} > 0$, otherwise 0

Global frequency

- Assumption: rare terms are more informative/important than frequent terms

- Consider the query „arachnocentric shop”
  - A document containing rare term „arachnocentric” is more likely to be relevant than the document containing the more frequent term „shop”
  - We want a higher weight for rare terms like „arachnocentric”

- We will use document frequency, i.e., the number of documents in the collection to account for global rarity/frequency of the terms

Inverse document frequency

- Assumption: the informativeness of the term *t* is inversely proportional to the number of documents in the collection in which the term appears
  - The less documents in which the term appears – the bigger weight

- Inverse document frequency (on the document collection D)

$$idf(t) = \log_{10}(|D| \;/\; |\{d' \in D : t \in d'\}|)$$

  - The logarithm is used to „dampen” the effect for terms that appear in very few documents
    - E.g., only in one or two documents
  - The base of the logarithm is not particularly important

Inverse document frequency – example

- Term frequency (TF) value of the term is computed for every document
  - N documents (d_1, d_2, ..., d_N) → N different TF scores for some term t_i
    - $tf(t_i, d_1)$, $tf(t_i, d_2)$, ..., $tf(t_i, d_N)$
- Inverse document frequency (IDF) is a single value for the term on the whole document collection D (does not depend on particular document)
    - $idf(t_i) = idf(t_i, D)$
- Example: N = 1 million documents

term	df(term)	idf(term)
Frodo	10000	2
Sam	1000	3
stab	100	4
the	1000000	1

- **Q:** What is the effect of idf for single-term queries?
- **A:** None. **Q:** Why?

Collection frequency vs. Document frequency

- **Collection frequency** is the total number of occurrences of the term in the entire collection, i.e., in all of the documents
  - I.e., counting multiple occurrences in documents
- Using (inverse) collection frequency could be an alternative to (inverse) document frequency

- **Q:** Which is better?
  - **Q:** Should „Frodo" or „blue" get a higher weight?

Word	Collection frequency	Document frequency
Frodo	100442	5135
blue	100350	20452

TF-IDF

- Finally, the weight for the term t_i within the document d_j is computed by multiplying the TF (local) and IDF (global) components:

$$w_{ij} = tf(t_i, d_j) * idf(t_i)$$

$$tf(t_i, d_j) = (1 + \log_{10}(f_{ti,dj})) \; / \; (1 + \log_{10}(\max\{f_{t',dj} : t' \in d_j\}))$$

$$idf(t_i) = \log_{10}(|D| \; / \; |\{d' \in D : t_i \in d'\}|)$$

- TF-IDF is the **best known weighting scheme in IR**
- TF-IDF score of term t within document d is larger
  - The larger the number of occurrences of t within d
  - The smaller the number of other documents d' in which t occurs

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Vector space model

- **Vector space model**
  - Documents and queries considered to be bags of words
  - Both documents and queries are represented as vectors of TF-IDF weights of vocabulary terms
    - TF-IDF score of vocabulary term not contained in the query/document is 0

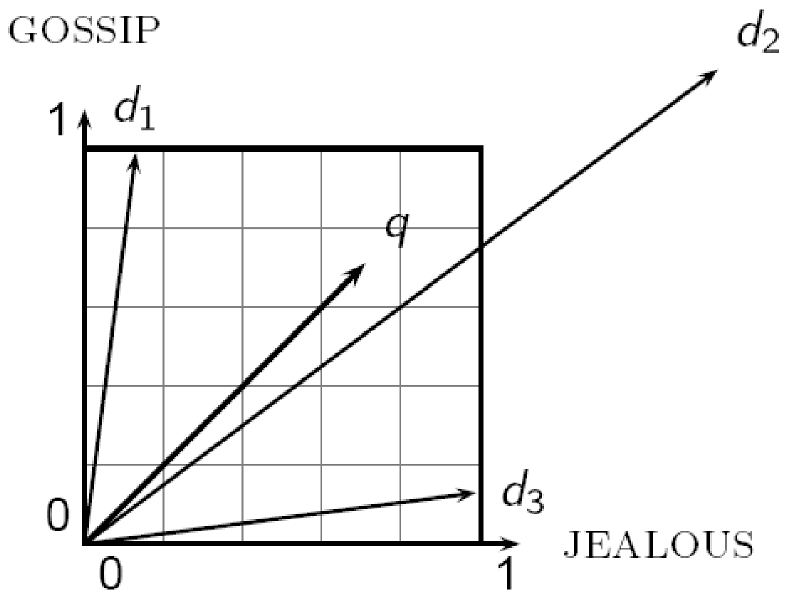
- Ranking function: similarity/distance between the two TF-IDF vectors (i.e., the vector of the document and the vector of the query)
  - **Q:** What distance metric to use?
    - Euclidean distance?
    - Any other distance/similarity metric?

Euclidean distance

- Euclidean distance
  - Measures the distance between the ends (points) of the two vectors

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- The Euclidean distance between q and d_2 is large
  - But the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar.
- E.g., q = [1, 2, 3, 4, 5], d_2 = [2, 4, 6, 8, 10]



Euclidean distance – shortcomings

- Take a document d and append it N times to itself – the obtained document is d’

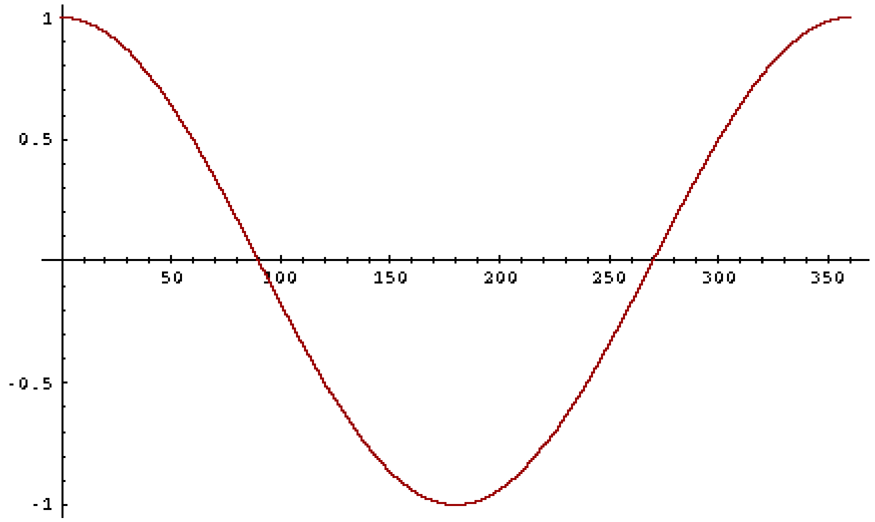
- Semantically, d and d’ have the same content
  - If N is large (i.e., we appended d to itself many times) the Euclidean distance between d and d’ is going to be large
  - Yet, d and d’ are semantically identical – d’ is as relevant for any query q as d is

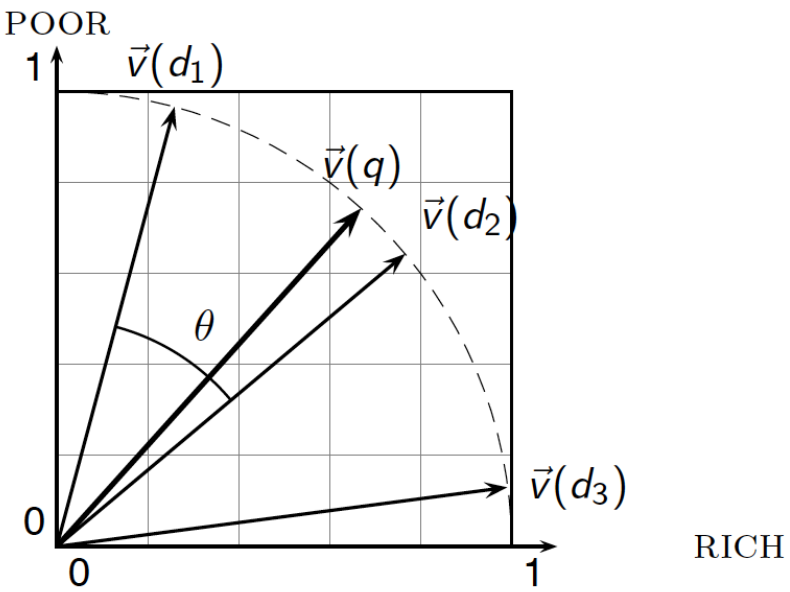
- However, the angle between vectors of d and d’ is going to be zero
  - These two vectors have exactly the same direction
  - Angle between the vectors better captures the actual similarity

- **Key idea:** rank documents according to the angle their vectors close with the vector of the query

Cosine similarity

- The smaller the angle between two vectors is, the larger is the value of the cosine of that angle
  - Cosine is a monotonically decreasing function on the [0°, 180°] interval





Cosine similarity

- **Cosine similarity** of two vectors is the cosine of the angle between them

$$cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

$$= \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{n} {x_i}^2} \cdot \sqrt{\sum_{i=1}^{n} {y_i}^2}}$$

- Cosine similarity is not affected by the length of the input vectors (norms in the denominator)
- Cosine distance d_C is simply computed as $d_C(\mathbf{x}, \mathbf{y}) = 1 - \cos(\mathbf{x}, \mathbf{y})$

Normalization of vector length

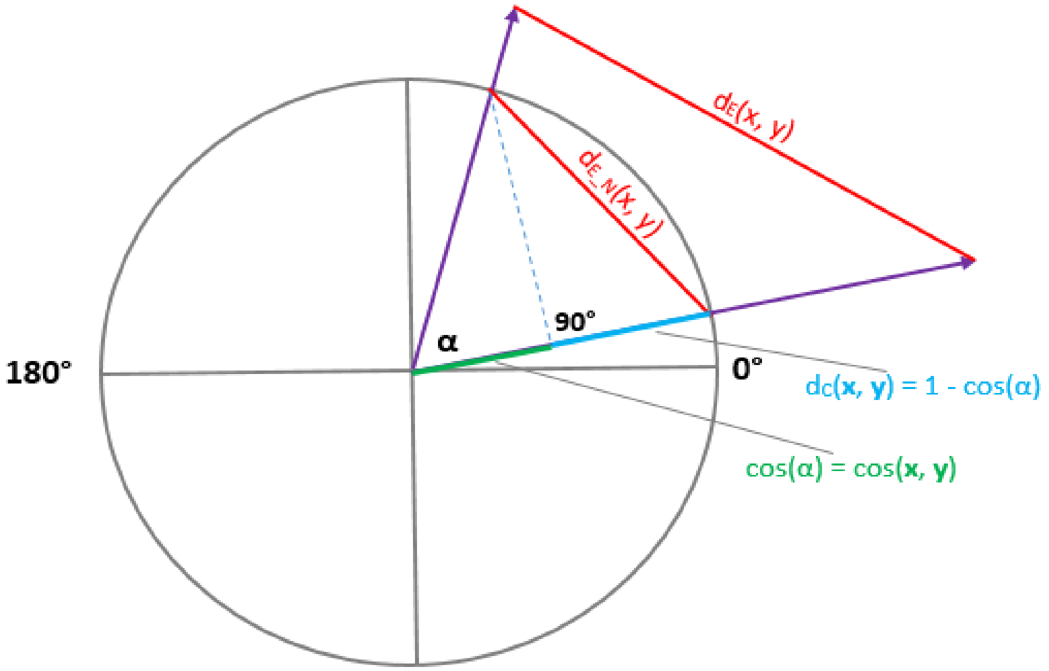
- **Q:** If the length is the issue for Euclidean distance, why don't we simply compute the Euclidean distance between unit-normalized vectors?
- **Q:** What is the relation between Euclidean distance of unit-normalized vectors and cosine distance?
- **A:** Cosine distance between two vectors is quadratically proportional to the Euclidean distance between unit-normalized versions of those vectors

$$d_C(\mathbf{x}, \mathbf{y}) = \frac{(d_{E_N}(\mathbf{x}, \mathbf{y}))^2}{2}$$

$$d_{E_N}(\mathbf{x}, \mathbf{y}) = d_E\left(\frac{\mathbf{x}}{\|\mathbf{x}\|}, \frac{\mathbf{y}}{\|\mathbf{y}\|}\right)$$

- **A:** The ranking produced by cosine distance is going to be the same as the ranking produced by Euclidean distance between unit-normalized vectors
- Cosine similarity between unit-normalized vectors amounts to their dot (scalar) product

Normalized Euclidean vs. Cosine distance



$$d_{E_N}(\mathbf{x},\mathbf{y}) = d_E\left(\frac{\mathbf{x}}{\|\mathbf{x}\|}, \frac{\mathbf{y}}{\|\mathbf{y}\|}\right)$$

$$= \left\|\frac{\mathbf{x}}{\|\mathbf{x}\|} - \frac{\mathbf{y}}{\|\mathbf{y}\|}\right\|$$

$$= \sqrt{\left(\frac{\mathbf{x}}{\|\mathbf{x}\|} - \frac{\mathbf{y}}{\|\mathbf{y}\|}\right)^T \cdot \left(\frac{\mathbf{x}}{\|\mathbf{x}\|} - \frac{\mathbf{y}}{\|\mathbf{y}\|}\right)}$$

$$= \sqrt{\frac{\mathbf{x}^T\mathbf{x}}{\|\mathbf{x}\|^2} - 2\frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} + \frac{\mathbf{y}^T\mathbf{y}}{\|\mathbf{y}\|^2}}$$

$$= \sqrt{1 - 2 \cdot cos(\mathbf{x},\mathbf{y}) + 1}$$

$$= \sqrt{2d_C(\mathbf{x},\mathbf{y})}$$

Vector space model – example

- Query: „Frodo stabs orc"
- Document collection
  - D1: „Frodo accidentally stabbed Sam and then some orcs"
  - d2: „Frodo was stabbing regular orcs but never stabbed super orcs – Uruk-Hais"
  - d3: „Sam was having a barbecue with some friendly orcs"

1. For all documents, compute the TF-IDF score for each query term

   idf(„Frodo") = $\log_{10}(3/2)$ = 0.176; tf(„Frodo", d1) = 1, tf(„Frodo", d2) = 1, tf(„Frodo", d3) = 0
   idf(„stab") = $\log_{10}(3/2)$ = 0.176; tf(„stab", d1) = 1, tf(„stab", d2) = 2, tf(„stab", d3) = 0
   idf(„orc") = $\log_{10}(3/3)$ = 0; tf(„orc", d1) = 1, tf(„orc", d2) = 2, tf(„orc", d3) = 1
   tf(„Frodo", q) = 1, tf(„stab", q) = 1, tf(„orc", q) = 1

2. Compute cosine similarities between vectors of q and each document
   - **Q:** Which term can we ignore for cosine similarity?
   - **Q:** Do we need to compute the norm of the query vector?

Alternative weighting and normalization schemes

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)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

- Most commonly implemented in IR systems:
  - TF: logarithmic, augmented, log average / log max – equally common
  - IDF: logarithmic
  - Normalization: L2 (Euclidian) norm (cosine similarity does it implicitly)
- Sometimes, the weighting schemes for query and documents may differ

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Speeding up retrieval with VSM

- Ranking all documents in the collection
  - Requires comparing the query TF-IDF vector with TF-IDF vectors of all documents
  - Infeasible for real-time querying on large collections

- We need to **reduce the cost** of cosine (dot product) computations
  1. By reducing the total number of cosines we compute
     a) Prefiltering candidate documents for ranking (e.g., via Boolean retrieval)
     b) By pre-clustering documents (based on their mutual similarity)
  2. By reducing the set of query terms we consider (e.g., according to IDF scores)
     - Smaller set of candidate documents
     - Faster cosine computation (shorter vectors for dot product)

Index-based elimination

- Documents that do not contain any of the query terms will have the cosine similarity of 0 with the query anyway
- Idea: Fetch only the documents that contain at least one query term
  - Using the inverted index
  - For the free text query „t_1 t_2 ... t_n” we spawn the Boolean query „t_1 OR t_2 OR ... OR t_n”
- Further possible speed-ups:
  1. Fetch only documents that contain more than N query terms
  2. Do not consider query terms with low IDF values
     - **Q:** Why?
     - **A:** Terms with low IDF scores appear in many (all?) documents in the collection, thus matching such terms between query and documents does not affect the ranking much
     - **A:** Posting lists of terms with low IDF are long – cosine computation for many documents

Pre-clustering documents

- If the document collection contains N documents, we randomly select $\sqrt{N}$ documents, which we call **leaders**
- For every other document in the collection
  1. Compute the similarities (cosine of the angle between TF-IDF vectors) with all leaders
  2. Add the document to the cluster of the most similar leader
- On average, a cluster will have $\sqrt{N}$ documents

- Random sampling of clusters is desirable (reflects the document distribution)
  - Faster than any other strategy for selecting leaders
  - Leaders reflect the data distribution
    - Dense regions will have more leaders than sparse regions

Pre-clustering documents

- Retrieval with document pre-clustering is much faster
  1. Measure the similarity of the query only with cluster leaders
     - $\sqrt{N}$ cosine computations
  2. Select the leader document d_L which is most similar to the query
  3. Compute the cosine similarities between the query vector and all documents in the selected leader's (d_L) cluster
     - $\sqrt{N}$ cosine computations
  4. (optional) if the users requires more results than there is documents in the cluster of the most similar leader d_L, proceed to the cluster of the next most similar leader
- With pre-clustering, total of $2\sqrt{N}$ cosine computations → $O(\sqrt{N})$
  - Quadratically lower complexity than before (without preclustering → $O(N)$)
- Shortcoming: pre-clustering may lead to lower recall
  - Some relevant documents may not be in the cluster of the most similar leader

Approximate cosine similarity

- Idea: reduce the length of the vectors on which we compute cosine similarity

- Only makes sense for queries with very many terms
  - If query has |V| terms, the cosine computation has complexity O(|V|)
  - Goal is to represent the query and document with a significantly shorter vector of length M, M << V
  - Cosine computation on lower dimensional vectors is then faster, O(M)

- Key question: how to select the lower-dimensional vector space in such a way that relations between the original cosine similarities are preserved?

Locality sensitive hashing

- A vector space of lower dimensions that (usually imperfectly) retains the distances from the original space is called a **low-dimensional embedding**
- Locality sensitive hashing (LSH)
  - A family of dimensionality reduction techniques that map the original vector space into a lower-dimensional space
  - Maximizing the extent to which the new vector space retains the topology of the original one
- One simple LSH method we will examine closer:
  - Random projections

Random projections

- A locality sensitive hashing method based on similarities with random vectors
- Hashing algorithm
  1. Choose a set of M random vectors $\{r_1, r_2, ..., r_M\}$ in the original high-dimensional vectors space (vector length $|V|$)
  2. For each document TF-IDF vector d do
     - Compute the inner (dot) product of d and each random vector r: $\theta(r, d) = \sum_i^{|V|} r_i * di$
     - Hash each inner product: $h(d, r_k) = 1$ if $\theta(r, d) > t$ (treshold), else 0
  3. Compute a new vector of hashes:
     - $d' = [h(d, r_1), h(d, r_2), ..., h(d, r_M)]$
     - The number of selected random vectors, M, is the dimensionality of hashed vectors
- **Q:** How does this hashing method preserve the relations between document distances of the original space?
  - If d_1 and d_2 are more similar than d_2 and d_3 in original space, why is it likely that d'_1 and d'_2 will be more similar than d'_2 and d'_3 in the projected space?

Champion lists

- For each term t_i store only the docs d_j with highest scores w_{ij}
  - I.e., Store only the documents for which this term is relatively informative
  - Since $idf(t_i)$ is the same for all documents, we rank documents according to the TF values, i.e., $tf(t_i, d_j)$
  - Put differently, if the term is relatively rare in the document, we treat it like it didn't appear in the document at all
    - Don't keep that document index in the term posting
- Such reduced term posting lists are called **champion lists** (aka *fancy lists*)
- The documents in the champion list can be decided in two different ways
  1. Taking the top N documents with highest $tf(t_i, d_j)$ scores
     - Posting lists of terms of same length N (unless the original posting was shorter)
  2. Taking all documents for which the $tf(t_i, d_j)$ is above some treshold value
     - Different lengths of postings for different terms

Champion lists

- Building the champion lists during indexing
  - Independent of any query that will be posed
  - When query is posed, it is possible that users wants more ranked results than what is the length of the champion list for some term
  - If champion lists are the only postings we kept, we cannot provide more results

- Solution: two-layer indexing
  - Champion lists and regular (full) posting lists

1. We try to answer the query using only the champion lists first
2. If the number of hits (= returned documents) using champion lists is smaller than the number of results user is looking for, return the hits using full posting lists

Tiered index

- Generalization of the two layer index
  - We can have posting lists of more than two layers (several segments)
- **Tiered index** is the index in which the postings are broken down hierarchically into several lists
  - Tiers of decreasing importance
  - For term t_i, break-down of documents is usually done according to the $tf(t_i, d)$ scores
  - In each tier, however, the documents are sorted according to docID, not $tf(t_i, d)$
    - We still need to perform posting merges in linear time
- Look-up in tiered index
  - We first look into the the top tier, i.e., merge the term postings of the first tier
  - If the merges over the top-tier postings result in too few hits, we continue to merge lists of the lower tiers

Tiered index – example

- „Frodo” -> T1: [2, 19, 24, 126]

  -> T2: [1, 3, 12, 27, 69, 111]

  -> T3: [7, 20, 76]
- „Sam” -> T1: [2, 18, 24, 158]

  -> T2: [1, 6, 69, 126]

  -> T3: [44, 90]
- Query: „Frodo and Sam”, we need to return at least 3 results!
  - Merge at T1: [2, 24] → only 2 results, we need to go to T2 as well
  - Second iteration
    - **Q:** merge(„Frodo”, „Sam”, T1) ∪ merge(„Frodo”, „Sam”, T2)?
    - **A:** No, we have to do – merge(sort(„Frodo”, T1, T2), sort(„Sam”, T1, T2))
    - Final result: [1, 2, 24, 69, 126]

Outline

- Recap of Lecture #3
- Ranked retrieval and scoring
- Vector space model
  - Term weighting (TF-IDF)
  - Ranking with cosine similarity
- Speeding up VSM retrieval
- Query parsing and multi-criteria ranking

Phrase queries and scoring function

- Remember the phrase queries from Lecture 2?
  - E.g., „Frodo Baggins”, „Las Angeles”, „hot potato”
- We handled the phrase queries with the positional index
- The vanilla vector space model uses the regular index
  - No positional information, pure bag-of-words document representation

- How can we account for phrase queries with VSM ranking?
  1. If proximity is a hard requirement from the users
     - Build the positional index and combine it with VSM ranking
  2. If proximity is a soft requirement (i.e., documents where query terms are closer together are preferred)
     - Incorporate a measure of query term proximity into a ranking function for documents
     - We still need the positional index ☹. **Q:** Why?

Query parsing and multiple query spawning

- IR systems often have query parsing components to analyse the queries
  - Based on the results of the analysis, the initial query can be „rewritten”
  - Some terms might be ommitted
- Your original query might not be the actual query to be matched against document collection
  - Your original query may be replaced with several queries
  - E.g., „rising interest rates” → „rising interest” and „interest rates”
- Example sequence of queries by query parser:
  1. Run the query as a phrase query „rising interest rates”
     - If enough hits, proceed to ranking
  2. If not enough hits in 1., spawn „rising interest” and „interest rates”
     - If enough hits, proceed to ranking of all documents fetched in 1. and 2.
  3. If still not enough hits, spawn „rising”, „interest”, and „rates”
     - Rank all retrieved documents in 1., 2., and 3. with VSM

Document quality

- Intuitive assumptions:
  - Documents have intrinsic quality which is independent of the queries being fired
    - E.g., more reliable (e.g., Wikipedia) vs. less reliable sources (spam sites)
  - In case when two documents have similar relevance for the query, we would like to rank one with higher quality above the one with lower quality
- **Static** document quality
  - Intrinsic property of the document itself, does not depend on other documents
  - E.g., digitally born documents have higher quality than OCR-ed ones
  - E.g., on the Web, we might consider Wikipedia pages to be of high quality
- **Dynamic** document quality
  - Depends on the associations with other documents
  - Link analysis based quality: crucial in web search (more in Lecture 11 ☺)

Aggregating different scores

- What if our ranking function needs to take into account several scores?
  - Cosine similarity of TF-IDF vectors
  - Proximity of query terms in documents
  - Static quality of documents

- Relevant questions:
  - What is the relative importance of different scores?
  - Are different scores even on the same scale (order of magnitude)?

- Methods
  - Expert designed aggregate function
  - **Learning to rank:** aggregate function learned with machine-learning algorithms
    - More in Lecture 9 ☺

Putting it all together

- Free text queries vs. Boolean queries (ranked retrieval vs. Boolean retrieval)
  - Query: „Frodo and Sam saw orcs"
  - Boolean: document relevant only if contains „Frodo" and „Sam" and „see" and „orc"
  - Ranked: document may be relevant if it, e.g., contains only „Frodo" and „orc"

- But the indexing mechanisms we introduced with Boolean retrieval are employed for ranked retrieval as well
  - Computing ranking scores for all documents is expensive
  - Using inverted index to obtain a smaller subset of documents, which are then ranked
    - But not too small – recall the tiered index

- We may have several different ranking criteria
  - We need to learn how to combine them into a single relevance score

After this lecture, you are...

- Are familiar with your first ranked retrieval model (VSM)
- Understand the TF-IDF term weighting scheme
- Know how to rank documents according to cosine similarity
- Know about some methods for speeding up VSM's ranking
- Are familiar with multi-criteria ranking