CS276A
Information Retrieval

Recap of the last lecture
- Parametric and field searches
  - Zones in documents
- Scoring documents: zone weighting
  - Index support for scoring
- $tf \times idf$ and vector spaces

This lecture
- Vector space scoring
- Efficiency considerations
  - Nearest neighbors and approximations

Documents as vectors
- At the end of Lecture 6 we said:
  - Each doc $j$ can now be viewed as a vector of $wf \times idf$ values, one component for each term
  - So we have a vector space
    - terms are axes
    - docs live in this space
    - even with stemming, may have 20,000+ dimensions

Why turn docs into vectors?
- First application: Query-by-example
  - Given a doc $D$, find others “like” it.
  - Now that $D$ is a vector, find vectors (docs) “near” it.

Intuition
Postulate: Documents that are “close together” in the vector space talk about the same things.
The vector space model

Query as vector:
- We regard query as short document
- We return the documents ranked by the closeness of their vectors to the query, also represented as a vector.

Desiderata for proximity
- If \( d_1 \) is near \( d_2 \), then \( d_2 \) is near \( d_1 \).
- If \( d_1 \) near \( d_2 \), and \( d_2 \) near \( d_3 \), then \( d_1 \) is not far from \( d_3 \).
- No doc is closer to \( d \) than \( d \) itself.

First cut
- Distance between \( d_1 \) and \( d_2 \) is the length of the vector \( |d_1 - d_2| \).
  - Euclidean distance
- Why is this not a great idea?
  - We still haven’t dealt with the issue of length normalization
    - Long documents would be more similar to each other by virtue of length, not topic
    - However, we can implicitly normalize by looking at angles instead

Cosine similarity
- Distance between vectors \( d_1 \) and \( d_2 \) captured by the cosine of the angle \( \theta \) between them.
  - Note – this is similarity, not distance
    - No triangle inequality for similarity.

Cosine similarity
- A vector can be normalized (given a length of 1) by dividing each of its components by its length – here we use the \( L_2 \) norm
  \[
  \| \mathbf{x} \|_2 = \sqrt{\sum x_i^2}
  \]
- This maps vectors onto the unit sphere:
  - Then, \( \| \mathbf{\tilde{x}} \| = \sqrt{\sum w_{ij}^2} = 1 \)
  - Longer documents don’t get more weight

Cosine similarity
\[
sim(d_j, d_l) = \frac{\mathbf{d}_j \cdot \mathbf{d}_l}{\| \mathbf{d}_j \| \| \mathbf{d}_l \|} = \frac{\sum_{i=1}^{n} w_{ij} w_{lk}}{\sqrt{\sum_{i=1}^{n} w_{ij}^2} \sqrt{\sum_{k=1}^{n} w_{lk}^2}}
\]
- Cosine of angle between two vectors
- The denominator involves the lengths of the vectors.

Normalization
Normalized vectors

- For normalized vectors, the cosine is simply the dot product:
  \[ \cos(\vec{d}_j, \vec{d}_k) = \vec{d}_j \cdot \vec{d}_k \]

Cosine similarity exercises

- **Exercise:** Rank the following by decreasing cosine similarity:
  - Two docs that have only frequent words (the, a, an, of) in common.
  - Two docs that have no words in common.
  - Two docs that have many rare words in common (wingspan, tailfin).

Exercise

- Euclidean distance between vectors:
  \[ |d_j - d_k| = \sqrt{\sum_{i=1}^{n} (d_{i,j} - d_{i,k})^2} \]
- Show that, for normalized vectors, Euclidean distance gives the same proximity ordering as the cosine measure.

Example

- **Example**
  - Docs: Austen's *Sense and Sensibility*, *Pride and Prejudice*; Bronte's *Wuthering Heights*
  - **Summary:**
    - **SaS** 115 58 20
    - **PaP** 10 7 11
    - **WH** 2 0 6
  - **Cosine similarity calculations:**
    - \( \cos(SaS, PaP) = 0.996 \times 0.993 + 0.087 \times 0.120 + 0.017 \times 0.0 = 0.999 \)
    - \( \cos(SaS, WH) = 0.996 \times 0.847 + 0.087 \times 0.466 + 0.017 \times 0.254 = 0.929 \)

Digression: spamming indices

- This was all invented before the days when people were in the business of spamming web search engines:
  - Indexing a sensible passive document collection vs.
  - An active document collection, where people (and indeed, service companies) are shaping documents in order to maximize scores.

Summary: What’s the real point of using vector spaces?

- **Key:** A user’s query can be viewed as a (very) short document.
- Query becomes a vector in the same space as the docs.
- Can measure each doc’s proximity to it.
- Natural measure of scores/ranking – no longer Boolean.
  - Queries are expressed as bags of words
- Other similarity measures: see [http://www.lans.ece.utexas.edu/~strehl/diss/node52.html](http://www.lans.ece.utexas.edu/~strehl/diss/node52.html) for a survey.
Interaction: vectors and phrases

- Phrases don’t fit naturally into the vector space world:
  - “tangerine trees” “marmalade skies”
  - Positional indexes don’t capture tf/idf information for “tangerine trees”
- Biword indexes (lecture 2) treat certain phrases as terms
  - For these, can pre-compute tf/idf.
- A hack: we cannot expect end-user formulating queries to know what phrases are indexed

Vectors and Boolean queries

- Vectors and Boolean queries really don’t work together very well
- In the space of terms, vector proximity selects by spheres: e.g., all docs having cosine similarity ≥0.5 to the query
- Boolean queries on the other hand, select by (hyper-)rectangles and their unions/intersections
- Round peg - square hole

Vectors and wild cards

- How about the query tan* marm*?
  - Can we view this as a bag of words?
  - Thought: expand each wild-card into the matching set of dictionary terms.
  - Danger – unlike the Boolean case, we now have tfs and idfs to deal with.
  - Net – not a good idea.

Vector spaces and other operators

- Vector space queries are apt for no-syntax, bag-of-words queries
  - Clean metaphor for similar-document queries
- Not a good combination with Boolean, wild-card, positional query operators
- But …

Query language vs. scoring

- May allow user a certain query language, say
  - Freetext basic queries
  - Phrase, wildcard etc. in Advanced Queries.
- For scoring (oblivious to user) may use all of the above, e.g., for a freetext query
  - Highest-ranked hits have query as a phrase
  - Next, docs that have all query terms near each other
  - Then, docs that have some query terms, or all of them spread out, with tf/idf weights for scoring

Exercises

- How would you augment the inverted index built in lectures 1–3 to support cosine ranking computations?
- Walk through the steps of serving a query.
  - The math of the vector space model is quite straightforward, but being able to do cosine ranking efficiently at runtime is nontrivial
Efficient cosine ranking

- Find the $k$ docs in the corpus “nearest” to the query $\Rightarrow k$ largest query-doc cosines.
- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the $k$ largest cosine values efficiently.
    - Can we do this without computing all $n$ cosines?

Computing a single cosine

- For every term $i$, with each doc $j$, store term frequency $tf_{ij}$.
  - Some tradeoffs on whether to store term count, term weight, or weighted by idf.
- At query time, accumulate component-wise sum
  \[ \text{sim}(\vec{d}_j, \vec{a}_k) = \sum_{i=1}^{m} w_{ij} \times w_{ik} \]
- If you’re indexing 5 billion documents (web search) an array of accumulators is infeasible.

Efficient cosine ranking

- What we’re doing in effect: solving the $k$-nearest neighbor problem for a query vector.
- In general, do not know how to do this efficiently for high-dimensional spaces.
- But it is solvable for short queries, and standard indexes are optimized to do this.

Encoding document frequencies

*Add $tf_{id}$ to postings lists*
- Almost always as frequency – scale at runtime
- Unary code is very effective here
- $\gamma$ code (Lecture 3) is an even better choice
- Overall, requires little additional space

Computing the $k$ largest cosines: selection vs. sorting

- Typically we want to retrieve the top $k$ docs (in the cosine ranking for the query)
  - not totally order all docs in the corpus
  - can we pick off docs with $k$ highest cosines?

Use heap for selecting top $k$

- Binary tree in which each node’s value $> \text{values of children}$
- Takes $2n$ operations to construct, then each of $k \log n$ “winners” read off in $2\log n$ steps.
- For $n=1M$, $k=100$, this is about 10% of the cost of sorting.
Bottleneck

- Still need to first compute cosines from query to each of \( n \) docs → several seconds for \( n = 1 \text{M} \).
- Can select from only non-zero cosines
  - Need union of postings lists accumulators (\(<1\text{M}) on the query aargh abacus would only do accumulators 1,5,7,13,17,83,87 (below).

\[
\begin{align*}
\text{aargh} & \quad 2 \quad 1,2 \quad 7,3 \quad 83,1 \quad 87,2 \\
\text{abacus} & \quad 8 \quad 1,1 \quad 5,1 \quad 13,1 \quad 17,1 \\
\text{acacia} & \quad 35 \quad 7,1 \quad 8,2 \quad 40,1 \quad 97,3
\end{align*}
\]

Removing bottlenecks

- Can further limit to documents with non-zero cosines on rare (high idf) words
- Enforce conjunctive search (a la Google): non-zero cosines on all words in query
  - Get # accumulators down to \{min of postings lists sizes\}
  - But still potentially expensive
    - Sometimes have to fall back to (expensive) soft-conjunctive search:
    - If no docs match a 4-term query, look for 3-term subsets, etc.

Can we avoid this?

- Yes, but may occasionally get an answer wrong
  - a doc not in the top \( k \) may creep into the answer.

Best \( m \) candidates

- **Preprocess:** Pre-compute, for each term, its \( m \) nearest docs.
  - (Treat each term as a 1-term query.)
  - Lots of preprocessing.
  - Result: “preferred list” for each term.
- **Search:**
  - For a \( t \)-term query, take the union of their \( t \) preferred lists – call this set \( S \), where \(|S| \leq mt\).
  - Compute cosines from the query to only the docs in \( S \), and choose the top \( k \).

\[
\text{Need to pick } m > k \text{ to work well empirically.}
\]

Exercises

- Fill in the details of the calculation:
  - Which docs go into the preferred list for a term?
  - Devise a small example where this method gives an incorrect ranking.

Cluster pruning: preprocessing

- Pick \( \sqrt{n} \) docs at random: call these leaders
- For each other doc, pre-compute nearest leader
  - Docs attached to a leader: its followers;
  - Likely: each leader has \( \sim \sqrt{n} \) followers.
Cluster pruning: query processing

- Process a query as follows:
  - Given query $Q$, find its nearest leader $L$.
  - Seek $k$ nearest docs from among $L$’s followers.

Visualisation

Why use random sampling

- Fast
- Leaders reflect data distribution

General variants

- Have each follower attached to $a=3$ (say) nearest leaders.
- From query, find $b=4$ (say) nearest leaders and their followers.
- Can recur on leader/follower construction.

Exercises

- To find the nearest leader in step 1, how many cosine computations do we do?
  - Why did we have $\sqrt{n}$ in the first place?
  - What is the effect of the constants $a,b$ on the previous slide?
  - Devise an example where this is likely to fail – i.e., we miss one of the $k$ nearest docs.
  - Likely under random sampling.

Dimensionality reduction

- What if we could take our vectors and “pack” them into fewer dimensions (say 50,000→100) while preserving distances?
- (Well, almost.)
  - Speeds up cosine computations.
- Two methods:
  - Random projection.
  - “Latent semantic indexing”.
Random projection onto $k << m$ axes

- Choose a random direction $x_1$ in the vector space.
- For $i = 2$ to $k$,
  - Choose a random direction $x_i$ that is orthogonal to $\{x_1, x_2, \ldots, x_{i-1}\}$.
- Project each document vector into the subspace spanned by $\{x_1, x_2, \ldots, x_k\}$.

E.g., from 3 to 2 dimensions

$\begin{pmatrix} x_1 \cdot d_1 \\ x_1 \cdot d_2 \end{pmatrix}$ \quad $\begin{pmatrix} x_2 \cdot d_1 \\ x_2 \cdot d_2 \end{pmatrix}$

$\begin{pmatrix} x_1 \cdot d_1 \\ x_1 \cdot d_2 \end{pmatrix}$ \quad $\begin{pmatrix} x_2 \cdot d_1 \\ x_2 \cdot d_2 \end{pmatrix}$

Dot product of $x_1$ and $x_2$ is zero.

Guarantee

- With high probability, relative distances are (approximately) preserved by projection.
- Pointer to precise theorem in Resources.

Computing the random projection

- Projecting $n$ vectors from $m$ dimensions down to $k$ dimensions:
  - Start with $m \times n$ matrix of terms $\times$ docs, $A$.
  - Find random $k \times m$ orthogonal projection matrix $R$.
  - Compute matrix product $W = R \times A$.
  - $j$th column of $W$ is the vector corresponding to doc $j$, but now in $k \ll m$ dimensions.

Cost of computation

- This takes a total of $kmn$ multiplications. **Why?**
- Expensive – see Resources for ways to do essentially the same thing, quicker.
- Question: by projecting from 50,000 dimensions down to 100, are we really going to make each cosine computation faster?

Latent semantic indexing (LSI)

- Another technique for dimension reduction
- Random projection was data-independent
- LSI on the other hand is data-dependent
  - Eliminate redundant axes
  - Pull together “related” axes – hopefully
    - car and automobile
- More on LSI when studying clustering, later in this course.
<table>
<thead>
<tr>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>- MG Ch. 4.4-4.6; MIR 2.5, 2.7.2; FSNLP 15.4</td>
</tr>
<tr>
<td>- Random projection theorem: Dasgupta and Gupta. An elementary proof of</td>
</tr>
<tr>
<td>Fast Monte-Carlo Algorithms for finding low-rank approximations. IEEE</td>
</tr>
</tbody>
</table>