CS3245

Indexing

Document parsing

Preprocessing techniques:
- Tokenization.
- Removal of punctuations, numbers and dates, URLs, and/or passwords.
- Stop word removal.
- Normalization: e.g., removing within-term periods and hyphens, accents, etc.
- Case folding.
- Lemmatization and/or stemming.
- Others: fixing spelling variations (e.g., British vs American English), synonyms, or transliteration variations.

Unigram model:
- Parses documents as an unordered collection of tokens (i.e., bag-of-words).
- Does not model word order.

n-gram model:
- Parses documents as sequences of n tokens $\rightarrow$ requires $|V|^n$ storage space (including unseen sequences comprising of seen words).
- Utilizes the Markov assumption to predict a given word from the $n-1$ previous words.
- **True n-gram**: $P(x_1, ..., x_m) = \prod_{k=1}^{m} P(x_k | x_{k-n+1}, ..., x_{k-1})$
  Unigram style: $P(x_1, ..., x_m) = \prod_{k=n}^{m} P(x_{k-n+1}, ..., x_k)$
  The true n-gram model is more sensitive to the size of the dataset.

Various smoothing techniques can be applied to eliminate 0s from language models:

add-k: $P(x_n|x_1, ..., x_{n-1}) = \frac{C(x_1,...,x_n)+k}{C(x_1,...,x_{n-1})+kV}$

interpolation*: $P(x_n|x_1, ..., x_{n-1}) = \sum_{i=1}^{n-1} \lambda_i P(x_n|x_{n-i}, ..., x_{n-1})$

Kneser-Ney*: $P(x_n|x_{n-1}) = \frac{\max\{C(x_{n-1},x_n)-\delta,0\}}{C(x_{n-1})} + \lambda(x_{n-1}) P_{\text{KN}}(x_n)$
where $P_{\text{KN}}(x_n) = \frac{|\{x:C(x,x_n)>0\}|}{\sum_{x'}|\{w:C(w,x')>0\}|}$
and $\lambda(x_{n-1}) = \frac{\delta}{C(x_{n-1})} \times |\{x' : C(x_{n-1}, x') > 0\}|$

Index types

General:
- **Term-document incidence matrix**: matrix of 0s and 1s indicating whether a document contains a term. E.g.,

	d_1	d_2	d_3
t_1	0	0	0
t_2	1	1	1

- **Inverted indexes**: adjacency list which stores a list of documents containing t, for each term t (e.g., "term" $\longrightarrow$ [doc_id$_x$, doc_id$_y$, ...]).
  1. Documents tokenized (and preprocessed, if necessary).
  2. Sequence of term entries (i.e., (term, docID) pairs) are generated.
  3. Pairs are first sorted by term, then by docID.
  4. Identical pairs are merged, into (term, docFreq) $\longrightarrow$ [doc_id$_x$, doc_id$_y$, ...].

Boolean:
- **Skip pointers**: for speeding up merging in `AND` queries (does not work for `OR`/`NOT`). E.g., suppose we have $P_1 = p_{1,1} \rightarrow ... \rightarrow p_{1,m}$ and $P_2 = p_{2,1} \rightarrow ... \rightarrow p_{2,n}$, with a skip pointer $p_{1,i} \longrightarrow p_{1,j}$, and we are currently at $p_{1,i}$ and $p_{2,k}$.
  - If $p_{1,j} \leq p_{2,k}$, then we can use to skip pointer and skip from $p_{1,i}$ to $p_{1,j}$.
  - If $p_{1,j} > p_{2,k} > p_{1,i}$, then we cannot skip and will need to check $p_{1,i+1}$.

  Costs: construction/storage of pointers, additional comparisons during merging.
  Heuristic: implement $\sqrt{\text{len(postings)}}$ evenly-spaced skip pointers.

Phrasal:
- **Biword indexes**: indexes every consecutive pair of terms in the text; phrase queries can then be processed as an `AND` query on biwords (e.g., "a b c" $\rightarrow$ "a b" `AND` "b c").
  Limitation: could lead to false positives (e.g., "b c a b" is a valid result).
- **Extended biword indexes**: indexes all extended biwords in the form NX*N (e.g., "A and B" $\rightarrow$ "A B"); phrase queries can then be processed by extracting their extended biwords.
  Limitation: fails to handle complex phrases with multiple nouns (e.g., "National University of Singapore).
- **Positional indexes**: stores the positions of t in each document, for each term t (e.g., "term" $\longrightarrow$ [doc_id$_x$: [pos$_{x,1}$, ..., pos$_{x,k}$], doc_id$_y$: [pos$_{y,1}$, ...], ...]).
  Limitation: storage costs, slower retrieval relative to biword indexes.

Wildcard: additional index which maps from query to *dictionary terms* (not docIDs).
- **Permuterm indexes**:
  - Adds an end marker $ to a word, and indexes all rotations of the word in the corpus (e.g., "ab" $\longrightarrow$ ["$ab", "a$b", "ab$"]).
  - Wildcard queries can then be processed by prefix search on the rotation with the * at the end (e.g., "b*a" $\rightarrow$ search on "ab*").

  Limitation: size of index blows up proportionally to the average word length.
- **k-gram indexes**:
  - Enumerates all k-grams occurring in any term, and maintain an inverted index from the k-grams to dictionary terms matching each k-gram (e.g., "abc" $\longrightarrow$ ["$a" : ["abc", ...], "ab" : ["abc", ...], "bc" : ["abc", ...]]).
  - Wildcard queries can then be processed as an `AND` query on the k-grams (e.g., "ab*" $\longrightarrow$ search on "$a" `AND` "ab").

  Benefit: generally faster and more space efficient than permuterm index.
  Limitation: false positives (cf. biword indexes), may require post-query filtering.

Index construction

- **Blocked sort-based indexing (BSBI)**

```
1: function BSBI-IndexConstruction
2:     n ← 0
3:     while all documents have not been processed do
4:         n ← n + 1
5:         block ← ParseNextBlock()
6:         BSBI-Invert(block)
7:         WriteBlockToDisk(block, f_n)
8:     MergeBlocks(f_1, ..., f_n → f_merged)
```

  1. *ParseNextBlock*: Parse documents into termID-docID pairs, and accumulates the pairs in memory until a block of some fixed size is full (the dictionary storing (term $\rightarrow$ termID) mappings is kept in memory).
  2. *BSBI-Invert*: Sort the termID-docID pairs of each block in memory, and collect all termID-docID pairs with the same termID into a postings list.
  3. *WriteBlockToDisk*: Store intermediate sorted results on disk.
  4. *MergeBlocks*: Merge all intermediate results into the final index (via binary or n-way merge; latter more efficient if decent-sized chunks are read and written).

  – Time complexity: $O(T \log T)$, where T = the number of termID-docID pairs.
  (−) Dictionary must be able to fit into memory.
  (−) Fixed block size, must be decided in advance.
- **Single-pass in-memory indexing (SPIMI)**

```
1: function SPIMI-Invert
2:     output_file = NewFile()
3:     dict = NewHash()
4:     while free memory available do
5:         token ← next(token_stream)
6:         if term(token) ∉ dict then
7:             postings_list = AddToDict(dict, term(token))
8:         else
9:             postings_list = GetPostingsList(dict, term(token))
10:        if full(postings_list) then
11:            postings_list = DoublePostingsList(dict, term(token))
12:        AddToPostingsList(postings_list, docID(token))
13:    sorted_terms ← SortTerms(dict)
14:    WriteBlockToDisk(sorted_terms, dict, output_file)
15:    return output_file
```

  1. *Lines 3-12*: Going as far as memory allows, process pairs/tokens (parsed from documents) and generate an index (i.e., dictionary + postings) from these tokens using hashing.
  2. *Lines 13-14*: Sort terms lexicographically and write out the index.
  3. Merge all intermediate results into the final index (via binary or n-way merge).

  (+) No need to keep term-termID dictionary in memory.
  (+) No need to wait for fixed-size blocks to be filled up.
  (+) Adapts to the availability of memory.
  (+) Faster than BSBI, due to only sorting dictionary terms (instead of pairs).
- **Distributed indexing**: similar to *MapReduce*.
  - **Master**: break up input document, assign each split to an idle parser.
  - **Parser**: read documents, write (term, docID) pairs into term partitions.
  - **Inverter**: collect all pairs for some partition, sort and write to posting lists.
  - *Map* [doc $\rightarrow$ list(k, v)]: map splits of input data to key-value pairs.
  - *Reduce* [(k, list(v)) $\rightarrow$ out]: group pairs by keys, collect values, write to storage.
- **Dynamic indexing**: for collections where new documents are added over time.
  - **Linear merge**: maintain two indices (main and auxiliary), merge auxiliary into main index whenever the former gets too large (i.e., exceeds some n). $O(T^2)$
  - **Logarithmic merge**: maintain a series of indices:
    - Z_0: in-memory index.
    - I_i: on-disk indices, each twice as large as the previous one.

    If Z_0 gets too large, write to disk as I_0 or merge with I_0 (if it exists) as Z_1. Recursively write/merge to $Z_1, Z_2, ...$ until a valid write to some I_k can be performed. $O(T \log T)$

Index compression

Heaps' law: $|V| = kT^b$, typically $30 \leq k \leq 100$ and $0.4 \leq b \leq 0.6$.
Zipf's law: $cf_i = K/i$, where cf_i rep. # occurrences of a term in the collection.
Dictionary compression: compressing trees with nodes (term, docFreq, postingsPtr).
- **Dictionary-as-a-string**: store dictionary as a string of characters, replace term with termPtr (pointing to position of term in the string).
- **Blocking**: store term lengths within string, termPtr to every k^{th} term on the string (e.g., "...1a2ab3abc5abcde...").
- **Front coding**: store only differences between terms (e.g., "...2a*b2◇bc4◇bcde...").

Postings compression:
- **Gap encoding**: store just the gaps between adjacent docIDs.
- **Variable byte encoding**: use the fewest bytes to store a gap; the first bit of each byte is used as the *continuation bit*.

Retrieval

Boolean retrieval

- `X AND Y` $O(n_x + n_y)$
  1. Locate `X` and `Y` in the dictionary, and retrieve their postings.
  2. Intersect/Merge the two postings (i.e., walk through both postings simultaneously, stopping immediately when we reach the end of one of the postings), keeping only common entries.
- `X OR Y` $O(n_x + n_y)$
  1. Locate `X` and `Y` in the dictionary, and retrieve their postings.
  2. Union/Merge the two postings, keeping all entries appearing in any of the postings.
- `NOT X` $O(|D|)$
  1. Retrieve the full list of documents.
  2. Locate `X` in the dictionary, and retrieve its postings.
  3. Merge the full list and the postings (i.e., walk through simultaneously, stopping when reaching the end of the postings), keeping entries which do not appear in the postings.

Heuristics for optimizing boolean retrieval:
- $\underset{i}{\texttt{AND}}(\texttt{X}_i)$ and $\underset{i}{\texttt{OR}}(\texttt{X}_i)$
  - Process in the order of increasing document frequency (i.e., size of postings).
- `X AND NOT Y`: perform both operations as a single unit. $O(n_x + n_y)$
  - Similar to `NOT X`, but referring to `X`'s postings instead of the full document list.
- `X OR NOT Y`: no possible optimization, evaluate `NOT Y` before `OR`. $O(|D|)$

Tolerant retrieval

Dictionary implementation:
- **Fixed-length arrays**
  (−) Space wastage for shorter words, insufficient space for longer words.
  (−) Slow lookup (i.e., linear scan needed, $O(V)$).
- **Hash tables**: each vocabulary term hashed to an integer.
  (+) Faster lookup (i.e., $O(1)$).
  (−) Not very tolerant (e.g., no way to find word variants, or to do prefix search).
  (−) Need to rehash everything if vocabulary keeps growing.
- **B-trees**: each vocabulary term stored at the leaf nodes, sorted lexicographically.
  (+) Allows for prefix/suffix search.
  (−) Slower lookup (i.e., $O(\log |V|)$).
  (−) Expensive rebalancing.

Wildcard queries: can be handled using dictionaries implemented as B-trees.
- **Prefix queries**: e.g., "XY*" $\longrightarrow$ return all words w in the range XY $\leq w \leq$ XZ.
- **Suffix queries**: e.g., "*XY" $\longrightarrow$ return all words YX $\leq w \leq$ YY in a dictionary for *reversed terms*, or use permuterm/k-gram indexes.
- Others: e.g., "X*Y" $\longrightarrow$ intersect postings for "X*" and "*Y", or use permuterm/k-gram indexes.

Spelling correction (for queries):

- **Isolated word**:
  - Use a lexicon (e.g., standard English dictionary, or the indexed corpus) to obtain correct spellings.
  - Use either of the following to detect which word is closest to a misspelling:
    - Minimum edit distance: expensive and slower.
$$E(i,j) = \min\{E(i,j-1)+1,$$
$$E(i-1,j)+1,$$
$$E(i-1,j-1)+(X[i] \neq Y[j])\}$$
    - n-gram overlap: faster, but favor longer terms.
    - Jaccard coefficient: $\mathrm{JC}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$.
- **Context-sensitive**:
  1. Retrieve dictionary terms close to each query term.
  2. Enumerate all possible resulting phrases with one word "corrected" at a time.
  3. Phrases ranked/decided based on heuristics (e.g., by the number of hits).
- **Soundex**:
  1. Retain first letter of the word.
  2. Change all occurrences of $\{A, E, I, O, U, H, W, Y\}$ to 0.
  3. Change letters to digits as follows:
     - $\{B, F, P, V\} \to 1$
     - $\{C, G, J, K, Q, S, X, Z\} \to 2$
     - $\{D, T\} \to 3$
     - $\{L\} \to 4$
     - $\{M, N\} \to 5$
     - $\{R\} \to 6$
  4. Repeatedly remove one out of each pair of consecutive identical digits.
  5. Remove all 0s.
  6. Pad with trailing 0s and return the first 4 positions.

Scoring and ranking

Scoring can be done via:

- **Jaccard coefficient**: tf and idf not considered.
- **tf-idf**: $\mathrm{tfidf}_{t,d} = \mathrm{termFreq} \times \mathrm{docFreq} = w_{t,d}$ (typically *lnc.ltc*).
  termFreq rep. # times that t occurs in d, and
  docFreq rep. # documents that contain t.
- **Cosine score**: $\cos(\vec{q},\vec{d}) = \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$

```
1:  function VectorSpaceRanking(q)
2:      Scores[N] = 0
3:      Length[N]
4:      for each query term t do
5:          w_{t,q} ← tfidf_{t,q}
6:          postings_list ← postings list for t
7:          for each pair (d, tf_{t,d}) in postings_list do
8:              Scores[d] += w_{t,d} × w_{t,q}
9:      for each d do
10:         Scores[d] = Scores[d]/Length[d]
11:     return top k components of Scores[]
```

termFreq	docFreq	normalization
n (natural): $\mathrm{tf}_{t,d}$	n (none): 1	n (none): 1
l (log): $1 + \log(\mathrm{tf}_{t,d})$	t (idf): $\log \frac{N}{\mathrm{df}_t}$	c (cos): $\frac{1}{\sqrt{w_1^2+\ldots+w_M^2}}$
a (aug): $.5 + \frac{.5 \times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf): $\max\left\{0, \log\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\right\}$	u (pivoted unique): $\frac{1}{u}$
b (boolean): $\left[\mathrm{tf}_{t,d} > 0\right]$		b (byte): $\frac{1}{\mathrm{char\ len}^k}, k < 1$
L (log avg): $\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{avg}_{t\in d}(\mathrm{tf}_{t,d}))}$		

Heuristics for speeding up ranking:

- Assume $w_{t,q} = 1$ for each $t \in$ query: speeds up computation of cosine similarity.
- Use *heaps* to get top k elements over *arrays*: $O(J \log J + k) \to O(J + k \log J)$.
- Heuristics to prune less relevant docs (i.e., *non-contenders*) from collection.
  1. **Index elimination**: ignore part of the index based on the query.
     (a) *High idf query terms only*: only accumulate scores for $t \in q$ with high idf (e.g., non-stopwords).
     (b) *Documents containing many query terms*: only compute scores for docs containing at least k out of $|q|$ terms.
  2. **Tiered indexes**: break postings into hierarchy of lists ranging from most to least important.
     (a) *Champion lists*:
       - Pre-compute for each $t \in$ dict, r docs with highest $w_{t,d}$ in t's postings.
       - At query time, only compute scores of docs in champion list.
     (b) *High and low lists*:
       - Maintain two posting lists (*high* and *low*) for each term.
       - At query time, traverse *high* list before *low* list.
     (c) *Tiered indexes*: generalize *high* and *low* lists into tiers.
  3. **Impact-ordered postings**: sort the postings list of each $t \in q$ by $w_{t,d}$, only compute scores for docs whose $w_{t,d}$ is high enough.
     (a) *Early termination*: when traversing t's postings, stop early after either (1) r docs, or (2) $w_{t,d}$ drops below some threshold.
     (b) *idf-ordered query terms*: consider the postings of each t by decreasing idf.
  4. **Cluster pruning**:
     - Pick $\sqrt{N}$ docs at random as *leaders*, pre-compute the nearest (b_1) leader(s) for other docs (by computing the dot product of their vectors).
     - At query time, first find the nearest (b_2) leader(s) L, then seek the k nearest docs from L's followers (including L itself).

Incorporating additional information

1. **Static quality scores**: $\mathrm{net\_score}(q,d) = \alpha g(d) + (1-\alpha)\cos(q,d)$ where $g(d)$ is a query-independent quality score for each document.
2. **Query term proximity**: bump up docs where query terms occur near each other.
   Steps: phrasal → biword → vector space → rank union by VSM + proximity.
3. **Parametric and zone indexes**: allowing search by metadata (e.g., author, title, etc.)

Evaluation

- **Precision**: $P = \frac{\#\text{ relevant docs retrieved}}{\text{total }\#\text{ docs retrieved}}$
  - **Precision-recall curve**: computing precision at different recall levels.
  - **Interpolated precision**: taking the max precision to the right of the current recall level.
- **Recall**: $R = \frac{\#\text{ relevant docs retrieved}}{\text{total }\#\text{ relevant docs}}$
- F_β: $F = \frac{(1+\beta^2)PR}{\beta^2 P + R} = \frac{1}{\alpha/P + (1-\alpha)/R}$
- **Kappa measure** for inter-judge agreement: $\kappa = \frac{P(\text{agree}) - P(E)}{1 - P(E)}$ where
  $P(E) = P(\text{non-relevant})^2 + P(\text{relevant})^2$; 0 = chance, 1 = total agreement.
- **A/B testing**: diverting a small proportion of traffic to evaluate a new system.

Query Refinement

- **Rocchio algorithm**: $\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}}\vec{d}_j$
- **Query expansion**: expand each query term t with its related words from a thesaurus/synset. Generally increases recall but decreases precision.

XML IR

VSM for XML IR

1. Take each text node and break it into multiple nodes, one for each word.
2. Extract lexicalized subtrees (i.e., subtrees with ≥ 1 vocabulary term).
3. Index paths ending in a single vocabulary term.
4. Define similarity by **context resemblance**: $C_R(c_q, c_d) = \frac{1+|c_q|}{1+|c_d|}$ if c_q can be transformed into c_d by inserting additional nodes, 0 otherwise.
   Final score for docs computed via **SimNoMerge**:

$$\mathrm{SimNoMerge}(q,d) = \sum_{c_k \in B}\sum_{c_l \in B} C_R(c_k, c_l)\sum_{t \in V} w(q,t,c_k)\frac{w(d,t,c_l)}{\sqrt{\sum_{\substack{c \in B \\ t \in V}} w(d,t,c)^2}}$$

XML IR evaluation

- **Component coverage**:
  - (E) Exact coverage
  - (S) Too small
  - (L) Too large
  - (N) No coverage
- **Topical relevance**:
  - (3) Highly relevant
  - (2) Fairly relevant
  - (1) Marginally relevant
  - (0) Non-relevant

$$\#(\text{relevant items retrieved}) = \sum_{c \in A} Q\big(\mathrm{rel}(c), \mathrm{cov}(c)\big), \text{ where}$$

$$Q(\mathbf{rel}, \mathbf{cov}) = \begin{cases} 1.00 & \text{if (rel, cov)} = 3E \\ 0.75 & \text{if (rel, cov)} \in \{2E, 3L, 3S\} \\ 0.50 & \text{if (rel, cov)} \in \{1E, 2L, 2S\} \\ 0.25 & \text{if (rel, cov)} \in \{1S, 1L\} \\ 0.00 & \text{if (rel, cov)} = 0N \end{cases}$$

Probabilistic IR

Classic models

- **Binary independence model**: $P(R|d,q) = P(R|\vec{x},\vec{q}) = \frac{P(\vec{x}|R,\vec{q})P(R|\vec{q})}{P(\vec{x}|\vec{q})}$
  ranking docs by RSV/odds (more efficient):

$$\mathrm{RSV}_d = \log\Big(O(R|\vec{x},\vec{q})\Big) = \sum_{\substack{t:x_t=1 \\ q_t=1}} \log\left(\frac{p_t(1-u_t)}{u_t(1-p_t)}\right)$$

  where p_t rep. prob. of term appearing in a relevant doc,
  and u_t rep. prob. of term appearing in a non-relevant doc.
- **Nonbinary models**, e.g., **Okapi BM25**:

$$\mathrm{RSV}_d = \sum_{t \in q} \log\left(\frac{N}{\mathrm{df}_t} \times \frac{(k_1+1)\mathrm{tf}_{t,d}}{k_1\left((1-b)+b(L_d/L_{\mathrm{ave}})\right)+\mathrm{tf}_{t,d}}\right)$$

Language models

- **Default model**: $P(d|q) = P(d)\prod_{\text{distinct } t \in q} P(t|d) = P(d)\prod_{\text{distinct } t \in q}\frac{\mathrm{tf}_{t,d}}{|d|}$
- **Mixture model**: $P(d|q) = P(d)\prod_{\text{distinct } t \in q}\big(\lambda P(t|M_d) + (1-\lambda)P(t|M_c)\big)$
  - High λ: more conjunctive; favors documents containing all query words.
  - Low λ (e.g., < 0.5): more disjunctive; favors queries without all query words and longer queries.

Crawling

Basic crawler algorithm:

1. Initialize queue with URLs of known seed pages.
2. Repeat:
   (a) Take URL from queue.
   (b) Fetch and parse page.
   (c) Extract URLs from page.
   (d) Add URLs to queue.
   (e) Call the indexer to index the page.

Key concerns of crawlers:

- **Politeness**: should only crawl on allowed pages of a website.
- **Duplicate detection**: must be able to avoid duplicated contents and URLs.
- **Scalability**
- **Spider traps**: webpages which could lead to infinite loops.

Link analysis

Page relevance can be judged by:

- **Anchor text**: text surrounding a hyperlink.
- **Endorsements**: better to have (1) more, (2) originating from important sources, and (3) exclusive.

PageRank idea:

1. Start at a random page.
2. At each step, follow one of the n links on that page (i.e., each with probability $\frac{1}{n}$).
   - When a node has no outlinks, teleport to a random page (with equal prob).
   - Else, with probability α (e.g., 10%), teleport to a random page and follow a random link on the page instead with probability $(1-\alpha)$.
3. Do this repeatedly, and use the *long-term visit rate* as the page's score.

PageRank algorithm:

1. Start with *any* probability distribution (e.g., $x = (10\ldots0)$).
2. Multiply x by increasing powers of A (i.e., the transition matrix) until the product xA^k looks stable.
3. Eventually, we have $xA^k \approx a$, and we obtain the long-term visit rates of each state.