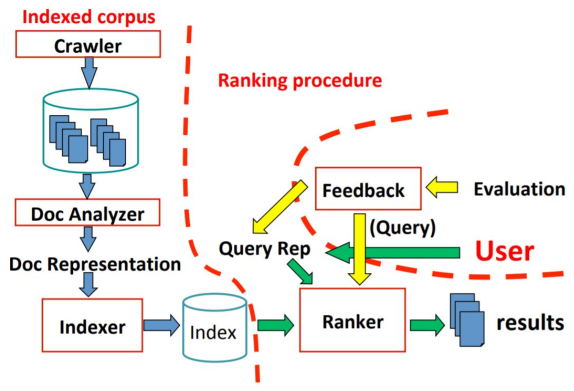
CSCE 670 Midterm Review Sheet

Web search basics

Three kinds of web search queries (from Broder)
✯ **Informational**: purely information needs. Acquire info in 1 or more pages.
✯ **Navigational**: straight heading to the site that I'd like to go (e.g. search "Facebook")
✯ **Transactional**: show me sites when I can perform certain transactions, like shop, download etc.
✯ Overall search engine architecture



Index corpus → Crawler → Doc analyzer (parsing, tokenization etc.) → Doc representation (typically BOW)
✲ **Web crawler**: (aka robots, spider, bot) an automatic program that systematically browses the web for the purpose of Web content indexing and updating
Complication: resources, malicious pages, duplicates, hitting servers too often... respect the robots.txt!
❄ **Visiting strategies**: BFS, DFS, focused crawling, etc.
❄ **Full text indexing**: Bag of Words (BOW) model. Mostly used! Assumption: words are independent from each other. Simple, but grammar and order are missing. Term-doc pairs.
Improvement on BOW: n-grams (a contiguous sequence of text - retain the sequences but ↑ in sizes), index docs with all the occurring words, etc.

❄ **Stopwords**: Useless words for query/document analysis, e.g. "the". !! If removed, would break the original meaning and structure of text
❄ **Normalization**: Convert different forms of a word to normalized form in the vocabulary, e.g. U.S.A → USA. Could be rule-based or dictionary-based.
❄ **Stemming**: Reduce inflected or derived words to their root forms, e.g. "ladies" → "lady". ✗ lose precise meaning of the word.
✜ **Alternative index structures**: **Skip pointers** - faster postings merges, **Positional index** - Phrase queries and Proximity queries, **Permuterm index** - Wildcard queries, **k-gram index** - Wildcard queries and spell correction

Modeling Text

✷ **TF**: term frequency. $tf_{t,d}$ = Number of occurrences of term t in document d. Determines the importance of a word by its occurrence in a document; neglects ordering, prefers frequent words.
The **log frequency weight** of term t in d is defined as $w_{t,d} = 1 + log_{10} tf_{t,d}$ if $tf_{t,d} > 0$, 0 otherwise. Better scaling of the raw TF.
✷ **IDF**: inverse document frequency. $idf_t = log\frac{N}{df_t}$, where df_t = the number of documents in the collection that contain a term t. It rewards rare terms. It's a measure of the informativeness of the term.
✷ **TF-IDF**: a combined weighting scheme. $tf-idf_{t,d} = tf_{t,d} \times idf_t$. It ↑ w/ the # of occurrences within a doc (TF), or with the rarity of the term in the collection (IDF).
❂ **Vector Space Model**: $\vec{V}(d)$ is the vector derived from document d, with one component in the vector for each dictionary term. Very high dim but sparse! (IR fundamental)
❂ **Cosine similarity**: the standard way of quantifying the similarity between two documents d_1 and d_2. Assume that vector representations are $\vec{V}(d_1)$ and $\vec{V}(d_2)$. $\therefore sim(d_1, d_2) = \frac{\vec{V}(d_1)\cdot\vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}$
Note: same idea for the cosine similarity could be applied to normalizing weights as well. Normalized weight = $\frac{ori_{weight}}{|Doc-vector|}$ (ori_{weight} could be tf, log tf, ...)

❂ **Zipf's law**: The i^{th} most frequent term has frequency cf_i proportional to $1/i$. That is, $cf_i \propto \frac{1}{i}$
❄ **BM25**: Handcrafted formula; some simple approximations to the 2-Poisson Model. (1 poisson captures the "general" words, 2 poisson captures those topic related/elite words)
→ para. in the 2-Poisson model unknown? Notice that the dist. saturated at some points from the graph. So approx. w/ a simple, saturated function as $\frac{tf}{tf+k}$. Use the B term $((1-b)+b\frac{|D|}{avgdl})$ to NORMALIZE THE LENGTH OF THE DOC. k and b selected heuristically.
Given a query Q, containing keywords $q_1, ..., q_n$, the BM25 score of a document D is: Score(D,Q) = $\sum_{i=1}^{n} IDF(q_i)\cdot\frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b\cdot\frac{|D|}{avgdl})}$
where $f(q_i, D)$ is q_i's term frequency in the document D, $|D|$ is the length of the document D in words, and *avgdl* is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

Evaluation

✎ **Test collections**: consist of 1. A document collection, 2. A test suite of information needs, expressible as queries, 3. A set of relevance judgments, standardly a binary assessment of either relevant or nonrelevant for each query-document pair.
✎ **Ground truth**: decision over a doc (rel or not).
✎ Relevance is assessed relative to an **information need** rather than a query. Info needs got translated into queries.
✎ **McNamara's fallacy**: don't measure easy stuffs but neglect the true objective
⋆ **CG**: cumulative gain. The CG at a particular rank position p is defined as: $CG_p = \sum_{i=1}^{p} rel_i$, where rel_i is the graded relevance of the result at position i. (unaffected by changes in the ordering of search results)
⋆ **DCG**: discounted cumulative gain. The DCG at a particular rank position p is defined as: $DCG_p = \sum_{i=1}^{p}\frac{2^{rel_i}-1}{log_2(i+1)}$. It penalizes rel doc showing up lower in the search result list. Also assuming highly rel docs are more useful.

⋆ **NDCG**: Normalized DCG. For a query, its NDCG at a particular rank position p is defined as: $NDCG_p = \frac{DCG_p}{IDCG_p}$, where IDCG = ideal discounted cumulative gain = $\sum_{i=1}^{|REL|} \frac{2^{rel_i}-1}{log_2(i+1)}$. This metric is used to compare a search engine's performance between queries (as queries might have different lengths).
❤ **Precision**: the fraction of relevant instances among the retrieved instances. = $\frac{tp}{tp+fp}$ = P(relevant|retrieved)
❤ **Precision@k**: measures precision but up to position k. Note that this is a ranked evaluation.
❤ **Recall**: the fraction of relevant instances that have been retrieved over the total amount of relevant instances. = $\frac{tp}{tp+fn}$ = P(retrieved|relevant)
There is an inverse relationship between precision and recall, which means that it's possible to optimize one at the expense of the other. (One could ↑ precision by returning more docs; recall is a non-decreasing function of the number of docs retrieved.)
❤ **F-measure**: a measure of a test's accuracy. It is defined as the weighted harmonic mean of the precision and recall of the test. F = $2 * \frac{precision \cdot recall}{precision + recall}$
Note: NDCG is better than precision and recall because it naturally incorporates **graded relevance** and order. Also precision/recall cannot distinguish duplicates; neither do them consider rank positions as well. Precision/recall rely on binary assessment, and they're both heavily skewed by corpus/authorship.

Link Analysis

PageRank

Idea: The importance of a page is judged by the number of pages linking to it as well as their own importance.
Original formula (w/o the random walker effect): $I = SI$, where I is the PR vector and S is a modified version of the transition matrix (captured portion of outlinks through each column; dangling node has prob. $1/n$).
Compute using the power method; initial vector x doesn't matter, but S needs to be an ergodic Markov chain in order for I to converge; it's aperiodic (so that the random walker can't visit pages sequentially) and irreducible (so that every page is connected to each other). Eventually, after k times, $xS^k = I$.
Google matrix: combine both the link structure & the random walker effect. (To escape from rank sink as well - assume that the surfer would get bored and teleport to other website) $G = \alpha S + (1-\alpha)\frac{1}{n}1$ (where 1 is an $n \times n$ matrix whose entries are all 1).
$\vec{x} = \vec{x} \cdot G$
PR = long-term visit rate

Topic-sensitive/Personalized PR

Non-uniform teleportation. Instead of randomly jumping to other pages, now all jump to the designated topic pages (e.g. 10% sports, 90% health, etc). Weighted sum of rank vectors. Could be computed either offline (so that each page would have multiple PR scores corresponding to diff. categories), or online (a dynamic PR score at runtime).

HITS

Starting point: root set (query → get results from a search engine) → base set (docs retrieved from the query + all their linking docs)
Iterative algorithm to compute hub and authority scores: (authority first then hub, initially $h(x) = a(x) = 1$)
✪ Authority: a direct answer to the information need. $a(x) = \sum_{y:y \to x} h(y)$ (sum of h(y), for all y pointing to x - inlinks) = $A^t\vec{h} = A^tA\vec{a}$
✪ Hub: a good list of links to pages answering the information need. $h(x) = \sum_{y:x \to y} a(y)$ (sum of a(y), for all y pointed to by x - outlinks) = $A\vec{a} = AA^t\vec{h}$
✪ Normalize after each iteration to scale down. Eventually converge! Only care about rel. order.
✪ Attacks on HITS: link to good authorities → becomes a hub → link to crappy pages → crappy pages now become good authorities
✷ BadRank: Midterm 2017 4c), also a paper about it (Kolda and Procopio 2009). A way to detect spams using link structure. If some pages are known to be bad, their badness should NOT be propagated through the outlinks BUT the inlinks. Basically reversing edges to penalize src of badness.
♣ PR and HITS, compare and contrast: 1) PR can be precomputed, while HITS has to be computed at query time. HITS too expensive. 2) PR and HITS make 2 diff. design choices concerning (i) the eigenproblem formalization (ii) the set of pages to apply the formalization to. 3) The 2 are orthogonal so can be applied together. 4) no discerning difference.

Basic ML

❑ **Manual classification**: used at the beginning of the web. ✓ Very accurate if job is done by experts; consistent when the problem size and team is small. ✗ Scaling manual classification is difficult and expensive.

❑ **Rule-based classification**: very complex rules, normally boolean. ✓ High accuracy with good rules; easy to implement; scales well ✗ cumbersome and expensive to build and maintain such a system.

❑ **Features**: a property of the training data/a column name in the training dataset. e.g. age, gender etc.

❑ **Train/validate/test split**: Training set is for training. Validation set is for tuning hyper parameters (aka "developmental" set). Test set is for testing. General split: 70/10/20. *Never ever touch your test data!*

❑ **K-fold CV**: Split the training data into k subsets, then train/test k times. Each time use 1 for validation and $k-1$ for training. CV is also a way to avoid overfitting.

❑ **Loss function**: measures how "bad" a system's prediction is in comparison to the truth. In particular, if y is the truth and $\hat{y}$ is the system's prediction, then $l(y, \hat{y})$ is a measure of error. (Regression: squared loss or absolute loss, binary/multiclass classification: zero/one loss. SVM: hinge loss)

❑ **Regularization**: controls the complexity of the function (to avoid overfitting per se)

Rocchio

Classical, simple; uses centroids to define the boundaries.

The centroid of a class c is computed as the vector average or center of mass of its members: $\vec{\mu}(c) = \frac{1}{|D_c|}\sum_{d \in D_c} \vec{v}(d)$. Do this on all the training examples.

Draw boundaries (hyperplanes). The boundary between two classes in Rocchio classification is the set of points with equal distance from the two centroids. Then for unseen x, classify it in accordance with the region it falls into.

! Does not guarantee that classifications are consistent with the training data!

KNN

A supervised, non parametric (no assumption of the underlying data dist.) and instance-based learning algorithm. Good for getting a pretty accurate classifier up and running in a short time w/o constraints on efficiency.

Given a positive integer K, an unseen observation x and a similarity metric d:

1. computes d between x and each training obs. through the whole dataset. Set A = K points in the training data that are closest to x.
2. estimates the conditional probability for each class: $P(y=j|X=x) = \frac{1}{K}\sum_{i \in A} I(y^{(i)} = j)$
3. assigns x to the class with the largest prob.

K is indeed a hyperparameter. (Basically the category of x is determined by its k^{th} nearest neighbors)

Increasing k will decrease variance and increase bias. Decreasing it will go the other way. If we increase k too much, then we no longer follow the true boundary line and we observe high bias. (tradeoff)

✓ No training necessary; accurate w/ large training data; linear training time

✗ Not accurate if not enough training data

Learning to Rank

❖ 3 types: pointwise, pairwise and listwise.

☞ Nallapati, using a classifier.

Setup: Define rel score $g(r|d,q) = w \times f(d,q) + b$.

SVM training goal: to have $g(r|d,q) \leqslant 1$ for nonrel doc and $g(r|d,q) \geqslant 1$ for rel doc.

SVM testing: decide relevant iff $g(r|d,q) \geqslant 0$

Experiments: 4 TREC data sets, use Lemur (a state-of-the-art open source IR engine) as the baseline & linear kernel (since it's comparable w/ the quadratic kernel). 6 features, all variants of tf, idf, and tf.idf scores.

Result: At best the results are about equal to Lemur. Easy to add more features!

☞ RankingSVM (to classify instance pairs as correctly ranked or incorrectly ranked): turns the oridinal ranking problem into classification in the pairwise space.

We want a ranking function f s.t. $c_i > c_k$ iff $f(\psi_i) > f(\psi_k)$. Suppose that $f(\psi_i) = w \cdot \psi_i$. Thus, $c_i > c_k$ iff $w \cdot (\psi_i - \psi_k) > 0$. Create a new instance space from such pairs: $\phi_u = \phi(d_i, d_k, q) = \psi_i - \psi_k, z_u = +1, 0, 1$ as $c_i >, =, < c_k$. We can build model over just cases for which $z_u = 1$.

☞ Cao's improvement on RankingSVM: + 2 more customization that are negelected in the original model.

1) Cost sensitivity: Correctly ordering the most relevant documents is crucial to the success of an IR system, while misordering less relevant results matters little. 2) Query normalization: If we treat all pairs of results for queries equally, queries with many results will dominate the learning.

→ Improved w/ a new loss function to address type of rank difference and size of ranked result set.

Results: test on the OHSUMED data set and MSN search show that the improved method outperforms RSVM.

Neural methods

♦ Background: need to understand the info need better. Ways to improve: use of anchor texts, relevance feedback, manual thesaurus, etc.

♦ Web-based kernel function: utilize the search engine. Issue query $x \rightarrow R(x)$ = set of n retrieved docs from $x \rightarrow$ Compute the TFIDF term vector v_i for each document $d_i \in R(x) \rightarrow$ Truncate each vector v_i to include its m highest weighted terms → C(x) be the centroid of the L2 normalized vectors $v_i \rightarrow$ $QE(x)$ = the L2 normalization of the centroid C(x). Given 2 short text snippets x and y, we define the semantic similarity kernel between them as: $K(x,y) = QE(x) \cdot QE(y)$

♦ word2vec: reduces the dim by going from big, sparse co-occurrence count vectors down to low dim word embedding.

Workflow: n-gram windows → tuple based training examples → shallow NN to learn the input embedding matrix w and output embedding matrix w' → softmax to map to prob. → compared with ground truth, fine-tuning through back propagation

Improvement: negative sampling, subsampling frequent words, hierarchical softmax to further reduce training cost

✓ Good performance, easy to train, able to show semantic meaning btwn words through vector algebra

♦ Neural Ranking Models with Weak Supervision: Use BM25 as an automatic tool to label the data (human labeling too expensive), comparing 3 ranking architectures (1 pointwise, 2 pairwise) + 3 input representations (dense/sparse vector, embedding), tested on Robust04 & ClueWeb, only rank+embed and rankprob+embed perform better than BM25.

Points: 1) goal should be learning the ranking; 2) feed all data to NN and let it learn itself, instead of feeding data w/ predefined features; 3) weak supervision works (in terms of training NN w/ massive amounts of data) but not beating the existing method (BM25 in this case).

Misc.