Feedback in Text Retrieval

- Relevance Feedback
  - Users make explicit relevance judgments on the initial results
    - judgments are reliable, but users don't want to make extra effort
    - Query -> retrieval engine -> results -> user -> judgements -> feedback
    - document collection -> retrieval engine
    - document collection -> feedback
    - feedback -> updated query -> retrieval engine
- Pseudo/Blind/Automatic Feedback
  - Top-k initial results are simply assumed to be relevant
    - judgements aren't reliable, but no user activity is required
- Implicit Feedback
  - User-clicked docs are assumed to be relevant; skipped ones non-relevant
    - judgements aren't completely reliable, but no extra effort from users

Feedback in Vector Space Model - Rocchio

- Feedback in vector space model
  - How can a TR system learn from examples to improve retrieval accuracy?
    - Positive examples: docs known to be relevant
    - Negative examples: doc known to be non-relevant
  - General methods; query modification
    - adding new (weighted) terms (query expansion)
    - adjusting weights of old terms
- Rocchio Feedback: Formula
  - Move query vector to the centroid of the relevant documents
  - want to move away from the negative documents
  - new query =
    - original query
    - average relevant documents (centroid)
    - average non-relevant documents (centroid)
    - alpha beta gamma control the amount of movement in the centroids
- Example of Rocchio Feedback
  - vector = {}
  - query = ()
  - centroid vector = () (average of each word in the vector using the weights)
  - centroid of the relevant or positive documents and the centroids of negative documents
  - new query for each term compute (alpha * 1 + Beta * positive centroid vector term - gamma * negative centroid term) repeat this for all the terms to obtain the new query vector
- Rocchio in Practice
  - Negative (non-relevant) examples are not very important
  - often truncate the vector (i.e., consider only a small number of words that have highest weights in the centroid vector) (efficiency concern)
  - avoid "over-fitting" (keep relatively high weight on the original query weights)
  - can be used for relevance feedback and pseudo feedback (beta should be set to a larger value for relevance feedback than for pseudo feedback)
  - usually robust and effective

Feedback in Text Retrieval - Feedback in LM

- Feedback in language models
  - Query likelihood method can't naturally support relevance feedback
  - Solution:

        - KL divergence retrieval model as a generalization of query likelihood
        - Feedback is achieved through query model estimation/updating
- KL Divergence Retrieval Model
    - basically generalize the frequency into a LM
    - basically the difference given by the probabilistic model given by what the user is looking for
    - Query Likelihood f(q,d) = summation c(w,q) * [log O_seen(w|d) / alpha_d (p(w|C)] + n log alpha_d
    - KL-divergence f(q,d) = summation [p(w|theta_q)log(p_seen(w|d) / alpha_d p(w|C)] + log alpha_d
    - p(w|theta_q) = c(w,Q) / |Q|
- Feedback as Model Interpolation
    - document D -> theta_d (estimate a document language model)
    - query q -> theta_q (estimate a query language model)
    - compute the KL-divergence D(query_q || query_d) -> results -> feedback documents
    - feedback documents -> theta_f (estimate a feedback language model)
    - theta_q’ = (1-alpha)theta_q + alpha(theta_f) -> theta_q
        - linear interpolation update model
        - alpha controls the amount of feedback
            - set 1 full feedback
            - set 0 no feedback
- Generative Mixture Model
    - P(source) ->_lambda (background words) P(w|C) -> w F = {d1 …. dn}
    - P(source) ->_1-lambda (topic words) P(w|theta) -> w F = {d1 …. dn}
    - log p(F|theta) = summation_i summation_w c(w;d_i)log[(1-lambda)p(w|theta) + lambdap(w|C)]
    - Maximum Likelihood theta_f = argmax_theta log p(F|theta)
    - lambda = noise in feedback documents
- Example of Pseudo-Feedback Query Model
    - Query: “airport security”
    - lambda = .9
    - lambda = .7
    - mixture model approach
- Summary
    - Feedback = learn from examples
    - Three major feedback scenarios
        - relevance, pseudo, implicit feedback
    - Rocchio for VSM
    - Query model estimation for LM
        - Mixture Model

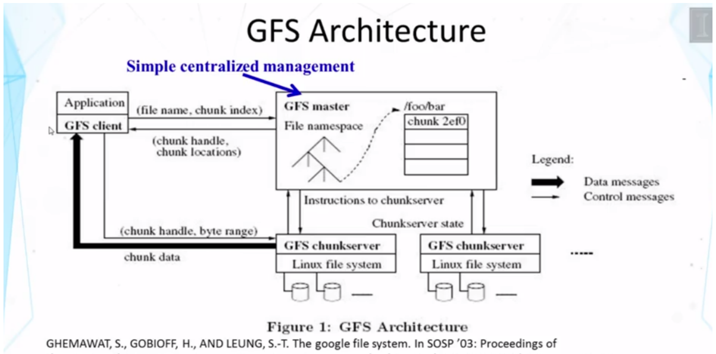
Web Search Introduction and Web Crawler

- Web Search: Challenges and Opportunities
    - Challenges
        - Scalability **Parallel indexing and searching (mapreduce)**
            - How to handle the size of the Web and ensure completeness of coverage?
            - How to serve many user queries quickly?
        - Low Quality information and spams **Spam detection and robust ranking**
        - Dynamics of the Web
            - New pages are constantly created and some pages may be updated very quickly

- Opportunities **Link analysis and multi-feature ranking**
  - many additional heuristics (e.g., Links) can be leveraged to improve search accuracy
- Basic Search Engine Technologies
  - Web -> crawler -> cached pages -> indexer -> (inverted) index <-> **retriever** -> results
    browser (user) -> query host info -> **retriever**
- Component 1: Crawler/Spider/Robot
  - Building a “toy crawler” is easy
    - start with a set of “seed pages” in a priority queue
    - fetch pages from the web
    - parse fetched pages for hyperlinks; add them to the queue
    - Follow the hyperlinks in the queue
  - A real crawler is much more complicated…
    - Robustness (server failure, trap, etc…)
    - Crawling courtesy (server load balance, robot exclusion, etc.)
    - Handling file types (images, PDF files, etc.)
    - URL extensions (cgi script, internal references, etc.)
    - Recognize redundant pages (identical and duplicates)
    - Discover “hidden” URLs (e.g., truncating a long url)
- Major Crawling Strategies
  - Breadth-First is common (balance server load)
  - Parallel crawling is nature
  - variation: focused crawling
    - targeting at a subset of pages
    - typically given a query
  - How to find new pages (they may not linked to an old page)
  - Incremental/repeated crawling
    - need to minimize resource overhead
    - can learn from the past experience (updated daily vs. monthly)
    - target at:
      - frequently updated pages
      - frequently accessed pages
- Summary
  - web search is one of the most important applications of text retrieval
    - new challenges: scalability, efficiency, quality of information
    - new opportunities: rich link information, layout, etc
  - crawler is an essential component of web search applications
    - initial crawling: complete vs. focused
    - incremental crawling: resource optimization

Web Indexing
- Overview of web indexing
  - standard IR techniques are the basis, but insufficient
    - scalability
    - efficiency
  - google’s contributions
    - google file system: distributed file system
    - MapReduce: software framework for parallel computation
    - Hadoop: Open source implementation of MapReduce
- GFS Architecture
  - Simple centralized management system

        - maintains file name space and lookup table



- MapReduce: A Framework for Parallel Programming
    - Minimize effort of programmer for simple parallel processing tasks
    - Features
        - Hide many low-level details (network, storage)
        - built-in fault tolerance
        - automatic load balancing
- MapReduce: Computation Pipeline
    - Input -> key value pairs
    - sent to map function
    - generates new key value pairs
    - MapReduce internal collection/sorting
    - Same keys are grouped together
    - Reduce(K,V[]) handles different key
    - processes the input key and set of values to produce another key and set of values to form the final output
- Word Counting
    - input: text data
    - output: count of each word
    - how can we do this within the MapReduce framework
- Word Counting: Map Function
    - Map(K,V) { For each word w in V, Collect(w,1); }
- Word Counting: Reduce Function
    - After internal grouping
    - Reduce(K,V[]) { Int count = 0; For each v in V, count += v; Collect(K,count); }
    - output the totals for the words - accumulate the counts
- Inverted Indexing with MapReduce
    - Map document 1 key value pairs, document 2 key value pairs
    - Built-In Shuffle and Sort: aggregate values by keys
    - Reduce the collection for each word. The counts from each document
- Inverted Indexing: Pseudo-Code

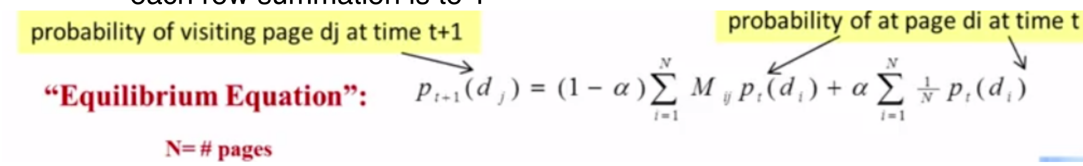
```
class Mapper
    procedure Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, posting ⟨n, H{t}⟩)
class Reducer
    procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
        P ← new List
        for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
            Append(P, ⟨a, f⟩)
        Sort(P)
        Emit(term t, postings P)
```

- Summary
  - Web scale indexing requires
    - storing the index on multiple machines (GFS)
    - creating the index in parallel (MapReduce)
- Both GFS and MapReduce are general infrastructures

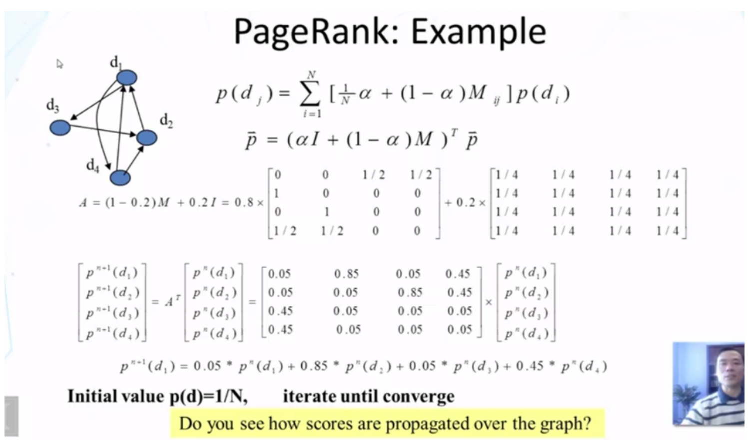
Link Analysis - Part 1

- Ranking algorithms for Web Search
  - Standard IR models apply but aren't sufficient
    - Different information needs
    - Documents have additional information
    - Information quality varies a lot
  - Major extensions
    - exploiting links to improve scoring
    - exploiting clickthroughs for massive implicit feedback
    - in general, rely on machine learning to combine all kinds of features
- Exploiting Inter-Document Links
  - Hub page
  - Authority page
- PageRank: Capturing Page "Popularity"
  - Intuitions
    - Links are like citations in literature
    - A page that is cited often can be expected to be more useful in general
  - PageRank is essentially "citation counting", but improves over simple counting
    - Consider "indirect citations"
      - being cited by a highly cited paper counts a lot…
    - smoothing of citations (every page is assumed to have a non-zero pseudo citation count)
  - PageRank can also be interpreted as random surfing
    - thus capturing popularity

Link Analysis Part 2

- The PageRank Algorithm
  - Random surfing model: At any page,
    - with prob. alpha, randomly jumping to another page
    - with prob. (1-alpha), randomly picking a link to follow
      - p(di): PageRank score of di = average probability of visiting page di
    - transition matrix
      - values indicating the probability of going from one page to another page
      - each row indicates a page
        - each row summation is to 1
- 

  - The first part captures the probability by following the link
  - The second part captures the probability be randomly jumping to another page
  - dropping the time index
  - p(d_j) = summation [1/n * alpha + (1-alpha) M_ij] p(d_j) -> p = ((aI + (1-alpha)M)^t) * p
  - We can solve this equation using iterative algorithm
- PageRank example



- PageRank in practice
  - Computation can be quite efficient since m is usually sparse
  - Normalization doesn't affect ranking, leading to some variants of the formula
  - The zero outlink problem: p(di)'s don't sum to 1
    - One possible solution = page-specific damping factor (alpha = 1.0 for a page with no out link)
  - Many extensions
  - Many other applications

Link Analysis Part 3

- HITS: capturing authorities and hubs
  - Intuitions
    - Pages that are widely cited are good authorities
    - Pages that cite many other pages are good hubs
  - The key idea of HITS (Hypertext-Induced Topic Search)
    - good authorities are cited by good hubs
    - good hubs point to good authorities
    - iterative reinforcement…
  - Many applications in graph/network analysis
- The HITS Algorithm
  - Adjacency matrix
- Summary
  - Link information is very useful
    - anchor text
    - pagerank
    - hits
  - both page rank and hits have many applications in analyzing other graphs or networks
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

- sdfsdf
-