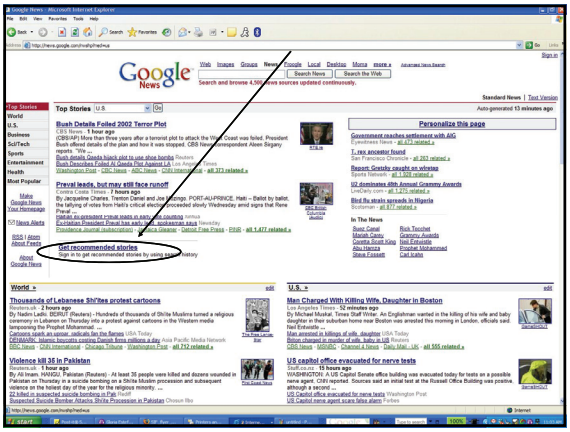
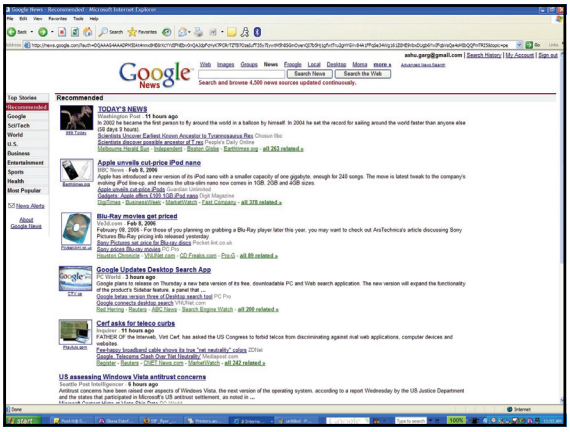
Google News Personalization

Most of the slides are due to Mayur Datar, thanks also to Slides by J. Eisner

Comments from the Class: Main characteristics

- Modify established machine learning algorithms for online setting.
- Importance of Scalability and fast response
- Solution: Separation of user data and story data
  - Offline processing of user similarities
  - Online processing of stories
- Domain independent system: not content based
- Empirically shown to be better than just recommending popular articles





Challenges

- Scale
  - Number of unique visitors in last 2 months: several millions
  - Number of stories within last 2 months: several millions
- Item Churn
  - News stories change every 10-15 mins
  - Recency of stories is important
  - Cannot build models ever so often
- Noisy ratings
  - Clicks treated as noisy positive vote

Approach

- Content-based vs. Collaborative filtering
- Collaborative filtering
  - Content agnostic: Can be applied to other application domains, languages, countries
  - Google's key strength: Huge user base and their data.
- Could have used content based filtering
- Focus on algorithms that are scalable

Algorithm Overview

- Obtain a list of candidate stories
- For each story:
  - Obtain 3 scores (y_1, y_2, y_3)
  - Final score = $\sum w_i * y_i$
- User clustering algorithms (Minhash (y_1), PLSI (y_2))
  - Score ~ number of times this story was clicked by other users in your cluster
- Story-story covisitation (y_3)
  - Score ~ number of times this story was co-clicked with other stories in your recent click history

Old memory-based approach

$$r_{u_a, s_k} = \sum_{i \neq a} I(u_i, s_k) w(u_a, u_i)$$

- Recommendation score based on other users click history and similarity measure to other users
  - Hence collaborative filtering
- However, this similarity matrix with size square of the number of the users is too large!

Their approach

$$r_{u_a, s_k} \propto \sum_{c_i : u_a \in c_i} \sum_{u_j : u_j \in c_i} I(u_j, s_k) w(u_a, c_i)$$

- Instead of n by n similarity matrix, compute only n by k similarity matrix, where k is the number of clusters.
- Also compute which clusters does each story belong in.

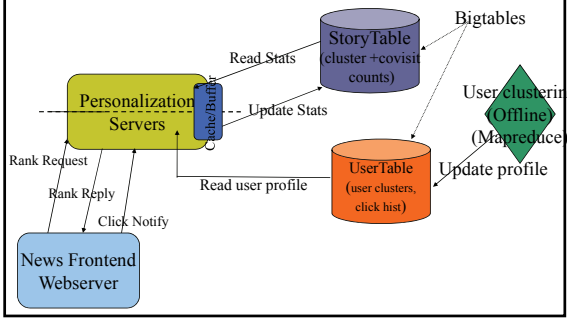
Algorithm Overview …

- User clustering done offline as batch process
  - Can be run every day or 2-3 times a week
- Cluster-story counts maintained in real time
- New users
  - May not be clustered
  - Rely on co-visitation to generate recommendations

Rest of the Talk

- System architecture
- Brief description of Minhash and PLSI
  - Mapreduce: making Minhash and PLSI scalable
- Experimental Results
  - Comparison to other algorithms
  - Live traffic evaluation

System Architecture



Rest of the Talk

- Exemplary system architecture
- Brief description of Minhash and PLSI
  - Mapreduce: making Minhash and PLSI scalable
- Experimental Results
  - Comparison to other algorithms
  - Live traffic evaluation

User clustering - Minhash

- Input: User and his clicked stories

$$S_u = \{s_1^u, s_2^u, \ldots, s_m^u\}$$

- User similarity = $\frac{|S_{u_1} \cap S_{u_2}|}{|S_{u_1} \cup S_{u_2}|}$
- Output: User clusters.
  - Similar users belong to same cluster

Minhash …

- Randomly permute the universe of clicked stories

$$\{s_1, s_2, \ldots, s_m\} \rightarrow \{s_1', s_2', \ldots, s_m'\}$$

- $MH(u) = \min(s_j^u)$ min defined by permutation

$$P\{MH(u_1) = MH(u_2)\} = \frac{|S_{u_1} \cap S_{u_2}|}{|S_{u_1} \cup S_{u_2}|}$$

- Treat MinHash value as ClusterId
- Use p different Minhashes, and put story s_k in cluster defined by $(MH_1(s_k), MH_2(s_k) \ldots MH_p(s_k))$
- Probabilistic clustering

Minhash...

- Implementation: Pseudo-random permutation
  - Compute hash for each story and treat hash-value as permutation index (instead of actually computing random permutations)
- Use map-reduce for calculation

Mapreduce

- Programmer specifies two primary methods:
  - map(k, v) -> <k', v'>*
  - reduce(k', <v'>*) -> <k', v''>*
- All v' with same k' are reduced together, in order.
- Usually also specify:
  - partition(k', total partitions) -> partition for k'
    - often a simple hash of the key
    - allows reduce operations for different k' to be parallelized

Example: Word Frequencies in Web Pages

- Input is files with one document per record
- Specify a *map* function that takes a key/value pair
  key = document URL
  value = document contents
- Output of map function is (potentially many) key/value pairs.
  In our case, output (word, "1") once per word in the document

"document1", "to be or not to be"

↓

"to", "1"
"be", "1"
"or", "1"
…

Example continued: word frequencies in web pages

- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The *reduce* function combines the values for a key
  In our case, compute the sum

key = "be" values = "1", "1"	key = "not" values = "1"	key = "or" values = "1"	key = "to" values = "1", "1"
↓	↓	↓	↓
"2"	"1"	"1"	"2"

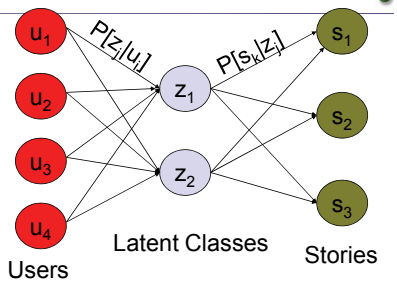
- Output of reduce paired with key and saved

"be", "2"
"not", "1"
"or", "1"
"to", "2"

MinHash as Mapreduce

- Map phase:
  - Input: key = user, value = story
  - Compute hash for each story (parallelizable across all data)
  - Output: key = cluster value = user
- Reduce phase:
  - Input: key = clusterid value = <list of users>

PLSI Framework



$P(s|u) = \sum_z P(s|z)*P(z|u)$

Background: LSI

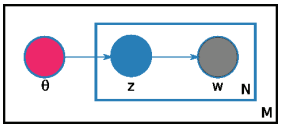
- Given a co-occurrence matrix between sets A and B, Latent Sematic Indexing produces a smaller set Z and relations between A-Z and Z-B.
  - i.e. Z is the latent class of "types", A is set of users, B is set of stories
  - example: {(car), (truck), (flower)} --> {(1.3452 * car + 0.2828 * truck), (flower)}
- Original LSI is based on SVD on the co-occurrence matrix
  - essentially dimension reduction

Background: LSI

- However, LSI may produce latent types that are hard to interpret:
  - {(car), (bottle), (flower)} --> {(1.3452 * car + 0.2828 * bottle), (flower)}
- Moreover, the probabilistic model of LSI does not match observed data
  - Assumes that A and B form a joint Gaussian, while a Poisson distribution is more reasonable

Background: PLSI

- **New Alternative:** PLSI
  - Assumes some prior distributions for relationships between A-Z and B-Z; uses Expectation Maximization to estimate the model
  - reported to give better results



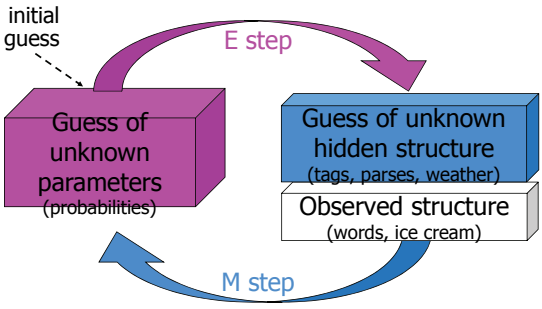
- PLSI for collaborative filtering **[Hofmann '04]**

Background: Expectation Maximization

- Well-known algorithm in statistics for finding maximum likelihood estimates of parameters in a probabilistic model, where the model depends on unobserved latent variables.
  - **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
  - **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

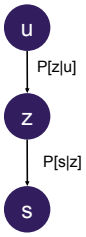
Repeat until convergence!

EM: General Idea



Clustering - PLSI Algorithm

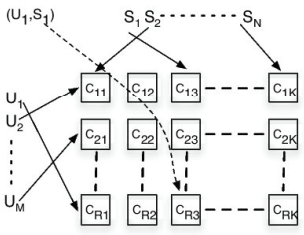
- Learning (done offline)
  - ML estimation
    - Learn model parameters that maximize the likelihood of the sample data
    - Ouput = $P[z_j|u]$'s $P[s|z_j]$'s
  - **$P[z_j|u]$'s lead to a soft clustering of users**
- **Runtime: we only use $P[z_j|u]$'s and ignore $P[s|z_j]$'s**



PLSI (EM estimation) as Mapreduce

- E step: (Map phase)
  $q^*(z; u,s,Ø) = p(s|z)p(z|u)/(\sum_z p(s|z)p(z|u))$
- M step: (Reduce phase)
  $p(s|z) = \sum_u q^*(z; u,s,Ø)/(\sum_s \sum_u q^*(z; u,s,Ø))$
  $p(z|u) = \sum_s q^*(z; u,s,Ø)/(\sum_z \sum_s q^*(z; u,s,Ø))$
- Cannot load the entire model from prev iteration in a single machine during map phase
- Trick: Partition users and stories. Each partition loads the stats pertinent to it

PLSI as Mapreduce



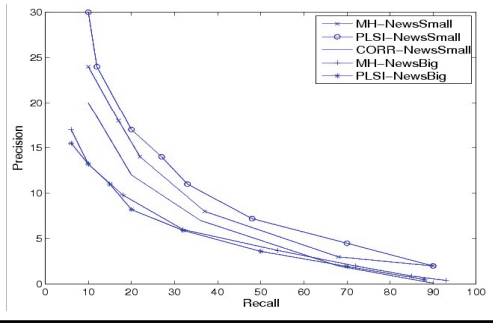
Covisitation

- For each story s_i store the covisitation counts with other stories $c(s_i, s_j)$
- Candidate story: s_k
- User history: $s_1, \ldots, s_n$
- score $(s_i, s_j) = c(s_i, s_j) / \sum_m c(s_i, s_m)$
- total_score$(s_k) = \sum_n$ score(s_n, s_k)
- Question from class: is this biased toward most popular news?

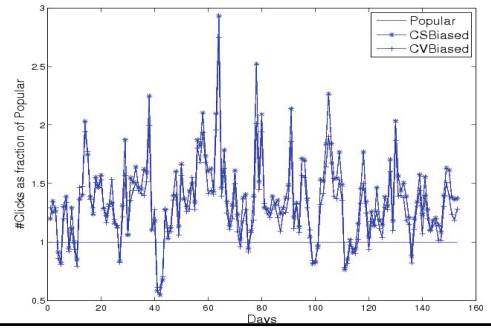
Rest of the Talk

- Exemplary system architecture
- Brief description of Minhash and PLSI
  - Mapreduce: making Minhash and PLSI scalable
- Experimental Results
  - Comparison to other algorithms
  - Live traffic evaluation

Experimental results



Live traffic clickthrough evaluation



Open Questions

- How to combine scores from different algorithms?
  - Linear combination seems to do worth than individual algorithms
  - Intuition: each algorithm is better in some cases and worse in others; So we should weight the algorithms depending on the case
  - Question from class: why not use individual algorithms if they do better?
- *Directional* co-visitation?

Other Comments from the Class

- PLSI's inability to handle dynamic data maybe solved using an approach similar to mini-batch?
- Challenge the assumption that users click what they care about:
  - Some people only look things up on Google news for which they have little knowledge
  - Connects with the paper's content that they worry about people not clicking news in subjects they know much about.

Thank You!