Learning Hostname Preference to Enhance Search Relevance

Jingjing Wang,[1] **Changsung Kang,**[2] **Yi Chang,**[2] **Jiawei Han**[1]
[1]University of Illinois at Urbana-Champaign
{jwang112, hanj}@illinois.edu
[2]Yahoo! Labs
{ckang, yichang}@yahoo-inc.com

Abstract

Hostnames such as *en.wikipedia.org* and *www.amazon.com* are strong indicators of the content they host. The relevant hostnames for a query can be a signature that captures the query intent. In this study, we learn the hostname preference of queries, which are further utilized to enhance search relevance. Implicit and explicit query intent are modeled simultaneously by a feature aware matrix completion framework. A block-wise parallel algorithm was developed on top of the Spark MLlib for fast optimization of feature aware matrix completion. The optimization completes within minutes at the scale of a million $\times$ million matrix, which enables efficient experimental studies at the web scale. Evaluation of the learned hostname preference is performed both intrinsically on test errors, and extrinsically on the impact on search ranking relevance. Experimental results demonstrate that capturing hostname preference can significantly boost the retrieval performance.

1 Introduction

Hostnames[1] can be high level indicators of the content they host. For example, the pages under *en.wikipedia.org* give people knowledge on an entity such as a concept, a product, a person, etc. For a query containing a knowledge seeking intent such as "what is roman art", a wiki page would be highly relevant. When a group of friends decide to go to Las Vegas for the Thanksgiving break, a simple query such as "vegas getaway" expects search results from *www.expedia.com*, *www.tripadvisor.com* or *www.orbitz.com*, etc.

In this paper, we study the hostname preference in the context of the fundamental search relevance problem. Modern search engines rely heavily on the relevance match between a query and an individual web page. If the search engines are also aware of the hostname preference of a query, so that they promote the search result pages from more relevant hostnames and demote the ones from less relevant hostnames, can we expect better retrieval performance out of this hostname level knowledge?

We investigate the query logs from Yahoo! search engine over one year. Hostname preference is learned by a feature-aware matrix completion model leveraging textual information, click information and popularity of the queries/hostnames. In our model, a query's preference for a hostname is determined by explicit intent matching (modeled by observed features), as well as latent intent matching (modeled by latent vectors). The learned preference is fed into a state-of-art machine learned ranking (learning-to-rank) system for investigation of its impact on search relevance.

In order to make the hostname preference learning actually work in commercial search engines, efficiency is crucial. Previous studies on matrix factorization report hours/days of training time even without considering features, which is impractical. We develop a block-wise distributed algorithm on top of the Spark Machine learning library(MLlib)[2], which completes within minutes at the scale of a million $\times$ million matrix. To the best of our knowledge, our work is the first to deal with large scale feature aware matrix completion in a block-wise distributed manner on hadoop clusters.

Contributions. We identify the novel yet fundamental problem of hostname preference learning for queries, and propose a preference learning model by feature aware matrix completion, modeling explicit and latent intent simultaneously. We develop a block-wise distributed algorithm for solving large scale feature aware matrix completion efficiently, which goes beyond the problem we consider and can be applied to a broad spectrum of applications such as large scale recommendation and clustering. Extensive experiments are performed on real-world data to investigate a) the impact of hostname preference on search ranking relevance, and b) the efficiency of our proposed algorithm.

2 Hostname Preference Learning Framework

2.1 Problem Formulation

We consider query logs from Yahoo! search engine over one year. The logs track for each query the URLs displayed as well as the URLs being clicked. We pre-process the query log to obtain a sequence of entries $\langle e_1, e_2, ... \rangle$, where each

[1]In this paper, a hostname is defined as the field before the first slash of a URL. We remove *https://* or *http://* for readability.

[2]http://spark.apache.org/docs/latest/mllib-guide.html

entry consists of a query q, a URL l, the number of clicks c at l for q, the number of views (or impressions, defined as the number of times the URL was shown to a user on the search page) v of l when q is issued: $e = (q, l, c, v)$. For each query q, the **hostname preference** is defined by a ranked list of hostnames with associated scores. A higher score indicates higher preference for the hostname.

Click Matrix. The $M \times N$ click matrix $\mathbf{Y}$, where M is the number of unique queries and N is the number of unique hostnames, represents the observed preference of queries over the hostnames. $\mathbf{Y}$ is a sparse matrix where the (i, j)th entry y_{ij} has the value of how many times the jth hostname has been clicked for the ith query. The number of clicks on a hostname for a query is computed by aggregating the clicks on the URLs which contain the hostname. Our goal is to approximate the click matrix $\mathbf{Y}$.

Intuition. In order to approximate the click matrix, a natural thought is to get vector representations of a query and a hostname, and use their inner product as the affinity score. We follow this general philosophy, and simultaneously consider observed features and latent factors, as well as the trade-off between them.

The query vector is viewed as an intent vector. Each dimension represents a certain intent. Meanwhile, a hostname vector shares this intent space and has a weight on each dimension. The inner product of these two vectors is an intent matching process. A larger value of the inner product indicates a higher preference. Explicit query intent can be captured by observed features such as keywords, while implicit query intent can be modeled as latent factors. In addition, we take into account the popularity of each query (measured by the total number of times the query is issued) and each hostname (measured by the total number of clicks on the hostname) to model the inherent bias of each entity.

2.2 Preference Model

The following generative process is devised for the click matrix[3] $\mathbf{Y}$. Our model complexity is intentionally restricted to provide insights into how the hostname level abstraction can benefit search relevance.

1. To generate the ith query q_i, $i = 1, 2, ..., M$,
   Latent Factors. Draw a latent intent vector $\mathbf{u}_i$ from the normal distribution: $\mathbf{u}_i \sim \mathcal{N}(\mathbf{0}, \sigma_i^q \mathbf{I})$.
   Features. Extract the explicit intent vector $\mathbf{u}'_i$ from observed information.
2. To generate the jth hostname h_j, $j = 1, 2, ..., N$,
   Latent Factors. Draw a latent intent vector $\mathbf{v}_j$ from the normal distribution: $\mathbf{v}_j \sim \mathcal{N}(\mathbf{0}, \sigma_j^h \mathbf{I})$.
   Features. Extract the explicit intent vector $\mathbf{v}'_j$ from observed information.
3. The number of click y_{ij} is determined jointly by
   a) How the latent intent of q_i and h_j match with each other: $\mathbf{u}_i^T \mathbf{v}_j$;
   b) How the explicit intent of q_i and h_j match with each other: $\mathbf{u}_i'^T \mathbf{v}'_j$;
   c) The popularity bias $pop(q_i), pop(h_j)$ and
   d) The weights $\mathbf{w}$ over different components. $\mathbf{w}$ is also generated by the normal distribution: $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma^w \mathbf{I})$.

   Thus we have $y_{ij} \sim \mathcal{N}(\mathbf{u}_i^T \mathbf{v}_j + \mathbf{w}^T \mathbf{f}_{ij}, \sigma^y)$, where $\mathbf{f}_{ij} = (\mathbf{u}_i'^T \mathbf{v}'_j, pop(q_i), pop(h_j))^T$.

$(\mathbf{Y}, \{\mathbf{f}_{ij}\})$ are the observations, $\{\boldsymbol{\sigma}^\alpha, \alpha = q, h, w, y\}$ is the set of hyperparameters and $\boldsymbol{\Theta} = (\{\mathbf{u}_i\}, \{\mathbf{v}_j\}, \mathbf{w})$ are the parameters to estimate. By maximum a posteriori (MAP) estimation, we obtain the following objective function:

$$(\{\mathbf{u}_i\}, \{\mathbf{v}_j\}, \mathbf{w}) = \arg\max_{\boldsymbol{\Theta}} P(\boldsymbol{\Theta} | \mathbf{Y}, \{\boldsymbol{\sigma}^\alpha\}) \quad (1)$$

$$= \arg\min_{(\{\mathbf{u}_i\}, \{\mathbf{v}_j\}, \mathbf{w})} \sum_{i,j} ||\mathbf{u}_i^T \mathbf{v}_j + \mathbf{w}^T \mathbf{f}_{ij} - y_{ij}||^2 + \sum_i \lambda_i^q ||\mathbf{u}_i||^2 + \sum_j \lambda_j^h ||\mathbf{v}_j||^2 + \lambda^w ||\mathbf{w}||^2$$

where $\lambda_i^q = \sigma^y / \sigma_i^q, \lambda_j^h = \sigma^y / \sigma_j^h, \lambda^w = \sigma^y / \sigma^w$.

Explicit Intent. Explicit intent vectors $\mathbf{u}'_i, \mathbf{v}'_j$ are constructed by investigating the keywords of queries.[4] We assign each keyword in a query to the URLs that are clicked when this query was issued, and aggregate all the keywords assigned to the URLs with the same hostname to be the associated keywords with that hostname. While a hostname can host tons of thousands of URLs and have diverse contents, the keywords associated with a hostname are fairly indicative and clean. Thus we characterize the explicit intent of a hostname by a multinomial term vector formed by its associated keywords.

The explicit intent of a query can be readily represented by its keywords. However, two similar queries such as "chinese buffet" and "all you can eat chinese" will have two quite different term vectors in this setting. To mitigate sparsity, we represent each query by a one-step propagation of the explicit intent from the clicked hostnames: the explicit intent vector of a query is defined as a weighted sum of the term vectors of its clicked hostnames. The weight for each hostname is proportional to the number of clicks.

Latent Intent. In addition to the explicit intent, we assume a low-rank latent space where the latent intent vectors of queries and hostnames reside in. The low-rank space captures a clustering effect where each cluster represents a latent intent. Intuitively, we can think of the latent intents as shopping, travel, job hunting, news seeking, etc.

2.3 Block-Wise Parallel Optimization for Large Scale Feature-Aware Matrix Completion

Our objective function (Eq. 1) naturally leads to a three-party alternating least squares (ALS) optimization problem as presented in Algorithm 1. While the algorithm itself is not mathematically sophisticated, efficient optimization at large scale is non-trivial. We focus our discussion on distributed algorithms on *Hadoop*, which is the most commonly used platform in modern search industry.

Numerous research efforts have been aimed at scaling up matrix factorization by parallelizing the updates of latent factors. Existing distributed algorithms exploit the independent update of the low rank vectors. However, the *shared* feature weights $\mathbf{w}$ in our problem makes these techniques for pure matrix factorization (with latent vectors only) not applicable (See Section 3 for a literature review), because $\mathbf{w}$ is affected by every entry in the matrix now.

To this end, we carefully examine the structure (additive feature integration) of our objective function and propose a block-wise distributed algorithm. It fuses the update of feature weights into the update of latent vectors, which introduces minimal overhead compared to distributed pure matrix factorization.

[3] We take the logarithm of each element y_{ij} and let $\mathbf{Y}$ be the click matrix after logarithm following the common practice for count data.

[4] Upon availability, the explicit intent can accommodate other features as well, associated with either a query, a hostname, or both.

Algorithm 1 A Three-Party ALS Algorithm for Feature Aware Matrix Completion

Input: A sparse click matrix $\mathbf{Y}$ with $(i,j) \in \Omega$ observed and the corresponding features $\{\mathbf{f}_{ij}\}$
Output: $\{\mathbf{u}_i\}, \{\mathbf{v}_j\}, \mathbf{w}$

Initialize $\{\mathbf{u}_i\}, \{\mathbf{v}_j\}, \mathbf{w}$
for *iter = 1:Maxiter* **do**
  Solve

$$(\lambda^w \mathbf{I} + \sum_{(i,j)\in\Omega} \mathbf{f}_{ij}\mathbf{f}_{ij}^T)\mathbf{w} = \sum_{i,j} \mathbf{f}_{ij} r_{ij}^w \quad (2)$$

  for $\mathbf{w}$, where $r_{ij}^w = y_{ij} - \mathbf{u}_i^T \mathbf{v}_j$.
  Solve

$$(\lambda_i^q \mathbf{I} + \sum_{j\in\mathcal{C}_i^q} \mathbf{v}_j \mathbf{v}_j^T)\mathbf{u}_i = \sum_{j\in\mathcal{C}_i^q} \mathbf{v}_j r_{ij}^u \quad (3)$$

  for $\{\mathbf{u}_i\}$, where $r_{ij}^u = y_{ij} - \mathbf{w}^T\mathbf{f}_{ij}$ and $\mathcal{C}_i^q = \{j|(i,j) \in \Omega,\ i.e., h_j \text{ is clicked by } q_i\}$.
  Solve

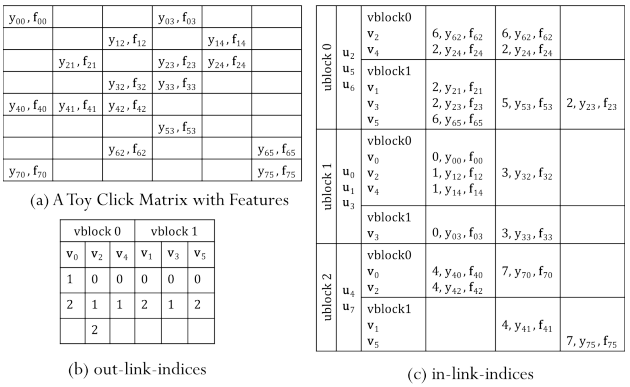
$$(\lambda_i^h \mathbf{I} + \sum_{i\in\mathcal{C}_j^h} \mathbf{u}_i \mathbf{u}_i^T)\mathbf{v}_j = \sum_{i\in\mathcal{C}_j^h} \mathbf{u}_i r_{ij}^v \quad (4)$$

  for $\{\mathbf{v}_j\}$, where $r_{ij}^v = y_{ij} - \mathbf{w}^T\mathbf{f}_{ij}$ and $\mathcal{C}_j^h = \{i|(i,j) \in \Omega,\ i.e., q_i \text{ has clicks at } h_j\}$.
end

Naive Solution. It's clear that Eq. (2) is a typical least square problem. A natural thought is to first solve Eq. (2) with fixed $\{\mathbf{u}_i\}$ and $\{\mathbf{v}_j\}$. The exact solution can be obtained by a single MapReduce job which gives $\sum_{i,j} \mathbf{f}_{ij}\mathbf{f}_{ij}^T$ and $\sum_{i,j} \mathbf{f}_{ij} r_{ij}^w$. Then with fixed $\mathbf{w}$, solving Eq. (3, 4) becomes a typical iteration in a pure matrix factorization. Each update of $\mathbf{w}$ requires a) one pass of the data to traverse $(\mathbf{Y}, \{\mathbf{f}_{ij}\})$ and b) retrieving $\{\mathbf{u}_i\}, \{\mathbf{v}_j\}$ to the corresponding nodes to compute $\{r_{ij}^w\}$. Being aware that global iteration is very expensive on hadoop than computation within each node, we propose to fuse the update of feature weights $\mathbf{w}$ into the update of the latent factors $\{\mathbf{v}_j\}$. *This reduces communication cost by a third and eliminates unnecessary data traversal.*

Infrastructure. Iterative algorithms with traditional Hadoop MapReduce is very time consuming since all the data including intermediate results (such as updated parameter values) are accessed via disks. We design our algorithm particularly for Spark since Spark provides primitives for in-memory cluster computing. This allows user programs to load data into a cluster's memory and query it repeatedly, making it well suited to iterative algorithms [Zaharia *et al.*, 2010]. A typical spark application is run on a driver node and multiple worker nodes, which fully supports traditional MapReduce jobs. For iterative algorithms, the driver node is naturally used to store the updated parameters and iteration counters, as well as to do central processing on the aggregated results such as the $\sum_{i,j} \mathbf{f}_{ij}\mathbf{f}_{ij}^T$ in our case. Each node has its local memory which can be used to persist its local data for fast access. Spark also supports shared variables across all nodes by an efficient broadcasting mechanism.

Algorithm Design. We build our algorithm on top of the blocked implementation of the ALS factorization algorithm in Spark MLlib. The basic idea is to update each $\mathbf{u}_i$ ($\mathbf{v}_j$) in Eq. (3) (Eq. (4)) in parallel only with the communication of the $\mathbf{v}_j$'s ($\mathbf{u}_i$'s) that are in $\mathcal{C}_i^q$ ($\mathcal{C}_j^h$). Meanwhile, we compute the statistics that are needed for the update of $\mathbf{w}$ together with the update of $\{\mathbf{v}_j\}$ so that we do not need an additional MapReduce job for updating $\mathbf{w}$.



(a) A Toy Click Matrix with Features

(b) out-link-indices

(c) in-link-indices

Figure 1: A Toy Example for Illustration of the Indices.
(b) out-link-indices for hostname blocks (vblock). It stores which query blocks should each $\mathbf{v}_j$ in this block be sent to.
(c) in-link-indices for query blocks (ublock). It stores which $\mathbf{u}_i$'s should the received $\mathbf{v}_j$'s from each hostname block update, together with the features $\mathbf{f}_{ij}$'s and click values y_{ij}'s needed for the update, where for each ublock b, $(i,j) \in \{(i,j)|\mathbf{u}_i \in \text{ublock } b\} \cap \Omega$.

The two sets of latent vectors ("queries $\{\mathbf{u}_i\}$" and "hostnames $\{\mathbf{v}_j\}$", respectively) are grouped into blocks. Two indices are created for each block: One *out-link-index* to index the destination blocks this block need to send its vectors to at each iteration; One *in-link-index* to index which vectors in this block need to be updated by the vectors sent from each source block. The subset of data ($\mathbf{Y}$ and $\{\mathbf{f}_{ij}\}$) corresponding to this block are also stored in the in-link-index to allow for local updates.

A toy click matrix is shown in Figure 1(a). We illustrate the indices for solving Eq. (3) using the in-link-indices of query blocks and the out-link-indices of hostname blocks as shown in Figure 1(b)(c). The weights $\mathbf{w}$ are broadcast to all the blocks at each iteration once it is updated. Then the indices allow for local update of each latent vector $\mathbf{u}_i$ in Eq. (3). Each hostname vector $\mathbf{v}_j$ is sent to the query blocks that need it for update. At most one copy of each hostname vector is sent to each query block at each iteration.

Now we explain how to get the statistics needed for updating $\mathbf{w}$ together with the update of $\{\mathbf{v}_j\}$. For each hostname block, once an update is to be performed, the corresponding query vectors are sent to the block. Based on the in-link-indices, each query vector will be used to update the corresponding hostname vector. After each update of $\mathbf{v}_j$, we have the latest $\mathbf{u}_i, \mathbf{v}_j$, together with the y_{ij} and $\mathbf{f}_{ij}$ from the in-link-indices, from which we can get the corresponding r_{ij}^w right away. Therefore we can obtain the local aggregate statistics of $\sum_{i,j} \mathbf{f}_{ij}\mathbf{f}_{ij}^T$ and $\sum_{i,j} \mathbf{f}_{ij} r_{ij}^w$ at each hostname block. Since the in-link-indices from all the hostname blocks together recover exactly the click matrix, we can then reduce the results from all hostname blocks to get $\sum_{i,j} \mathbf{f}_{ij}\mathbf{f}_{ij}^T$ and $\sum_{i,j} \mathbf{f}_{ij} r_{ij}^w$, and update $\mathbf{w}$ at the driver node.

Analysis of Memory Requirement and Complexity. We do not have the limitation that $\{\mathbf{u}_i\}$ or $\{\mathbf{v}_j\}$ must fit into each node's memory since the vectors are grouped into blocks.

The total in-link-index size for either queries or hostnames is slightly larger than the size of the data (clicks and features). In fact, we distribute two copies of the original data onto multiple nodes in the form of in-link-indices. During the optimization, all the indices are persisted in the local memory of the worker nodes, which guarantees efficient access.

The total out-link-index size is at most $O(B^q \times M + B^h \times N)$ where B^q is the number of query blocks and B^h is the number of hostname blocks. This space cost is much smaller than the in-link-indices.

At each iteration, each query vector $\mathbf{u}_i$ is sent at most once to each hostname block and vice versa. The communication cost at each iteration is at most $O(K(B^h \times M + B^q \times N))$ where K is the rank of the latent intent space (the length of a latent vector).

Since we fuse the computation of $\{r_{ij}^w\}$ to the update of $\{\mathbf{v}_j\}$, we naturally get the training error at each node. Thus the training error is obtained at each iteration together with the computation of the statistics needed for $\mathbf{w}$ update with no additional effort, which facilitates the convergence analysis of our model.

3 Related Work

Query Intent Learning. One line of research [Li *et al.*, 2008; Hu *et al.*, 2009; Diaz, 2009; König *et al.*, 2009] use classification to decide query intent, such as shopping, job hunting, news seeking, travel, etc. The classification itself serves as the end application (e.g. to determine if a news vertical should be shown). Other studies [Wu *et al.*, 2013; Ren *et al.*, 2014] consider query intent as intermediate results, which are used for query clustering or as features to improve search ranking relevance. Clicks, keywords and external knowledge such as wikipedia concepts are common information sources. Our framework can take advantage of the above finely calibrated models to learn either latent query intent or explicit query intent.

Website Recommendation. Song *et al.* [2011] study searchable web sites recommendation. While the searchable website resembles a similar level of abstraction of hostname, the scope is restricted to a small set of "searchable" websites, which heavily relies on the task specific features and does not involve latent factors. Ma *et al.* [2011] propose a collective matrix factorization model for personalized website recommendation, which investigates the click behavior of users, queries and URLs. The model is a pure click based Poisson loss factorization model where explicit features are not taken into account. The evaluation is only intrinsic on the test mean square/absolute error.

Search Ranking with Higher Level Information. Hostname preference is a higher level abstraction of individual URLs. Our preference learning framework provides a general recipe to study the impact on search ranking from various high-level factors. Other abstractions such as the category/topic/authority of a URL [Kleinberg, 1999; Gibson *et al.*, 1998; Xue *et al.*, 2005; Chandrasekar *et al.*, 2004] can be readily integrated into our framework as either explicit features or latent factors.

Large Scale Matrix Factorization.

Pure Matrix Factorization. Liu *et al.* [2010] parallelize the nonnegative matrix factorization using MapReduce as the low-rank vector representation of each entity can be updated in parallel. Zhou *et al.* [2008] and Teflioudi *et al.* [2012] propose distributed ALS algorithms with a similar philosophy. In these algorithms each computing node updates a subset of factors. Shared memory or a broadcasting scheme is used to make the updated factors available to all nodes. Yu *et al.* [2012] propose a parallel coordinate descent method to solve matrix factorization by optimizing one variable at a time instead of one vector. Gemulla *et al.* [2011] propose a distributed sochastic gradient descent (DSGD) method for matrix factorization, which examines the dependency of each update and specify a smart order of updates that can be done in parallel with MapReduce. Later Teflioudi *et al.* [2012] propose a DSGD++ algorithm which improves DSGD by using a new data partitioning and stratum schedule, and exploiting direct memory access and multi-threading.

Feature Aware Matrix Completion. Scaling up feature aware matrix completion is less explored due to the fact that all the entries in a matrix are dependent on the feature parameters. It is difficult to decouple the dependency to enable parallelization. Khanna *et al.* [2013] propose a "divide and conquer" algorithm for logistic loss matrix completion for binary response. The matrix to be approximated is partitioned into multiple sub matrices and the model is learned separately on each partition. The final factors are obtained by averaging the individual models. Performance of this algorithm is subject to the partition strategy and there is no guarantee for convergence. Shang *et al.* [2014] propose a parallel algorithm for matrix completion with multiplicative feature integration which applies iterative relaxation of the objective to facilitate parallel updates of parameters in a multi-threading setting. Our proposed algorithm is designed for a distributed environment with additive feature integration and it is guaranteed to converge to the local optimum.

4 Experiments

We report results from an intrinsic evaluation of the test root mean square error (RMSE), an extrinsic evaluation of the impact of modeling hostname preference on search ranking relevance, as well as an efficiency study for our proposed distributed optimization algorithm.

Data. We consider the query logs for general search (as opposed to vertical search) from Yahoo! search engine over one year. The raw data contains $5,869,014,608$ entries (*i.e.*, (q, l, c, v) tuples as defined in Section 2.1). We aggregate the clicks to hostnames, remove the queries issued fewer than 20 times, remove the hostnames that are clicked fewer than 20 times, and obtain the click matrix $\mathbf{Y}$ with $1,344,389,977$ observed entries, which involve $2,984,688$ unique queries and $1,654,212$ unique hostnames.

4.1 Experiment Design

The click matrix $\mathbf{Y}$ can be very noisy due to a) a user may accidentally click at a page which is not relevant at all and b) a relevant page is not clicked simply because it is ranked at a low position and does not appear in the first page. Therefore

we introduce a filtering parameter which filters out the query-hostname pairs that have very low ratio of *# impressions of a hostname* to *query frequency*, in order to get a less noisy dataset (*confident dataset*). The intuition is straightforward: the click number is reliable only if the hostname has enough views for the query. Then we randomly assign 80% of this confident dataset to the training set and the remaining 20% to the test set.

The regularization parameter λ_i^q is set to be proportional to the size of $\mathcal{C}_i^q$ and λ_j^h proportional to $\mathcal{C}_j^h$ following [Zhou *et al.*, 2008]. With this scaling we found the model is not sensitive to the regularization. For the results reported in what follows, $\lambda_i^q = 0.01|\mathcal{C}_i^q|$, $\lambda_j^h = 0.01|\mathcal{C}_j^h|$ and $\lambda^w = 0.01$.

We examine the training and test RMSE with different filtering parameters, different ranks of the latent intent space, and compare to baseline models under these different configurations. For the search ranking experiment, we use the entire confident dataset for training. The trained model is used to predict all the entries in $\mathbf{Y}$. The larger the filtering parameter, the more confident we are on our training set at the cost of a lower coverage.

4.2 Intrinsic Evaluation - RMSE

RMSE at Different Ranks for Different Model Settings

We plot the training and test RMSE with the filtering parameter 0.5 in Figure 2 considering the following models: 1) Pure regression model with the explicit features. 2) Pure factorization model with the latent intents. 3) Our proposed model: joint model of both explicit intent features and latent intent vectors. We investigate two initialization cases: a) $\mathbf{w}$ *update first.* The first update of the weights $\mathbf{w}$ is by the pure regression results. Then we update $\{\mathbf{u}_i\}$ with this $\mathbf{w}$ and $\{\mathbf{v}_j\}$ initialized by the standard multivariate normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$; and b) $\mathbf{w}$ *update last.* Initialize $\mathbf{w}$ to $\mathbf{0}$, $\{\mathbf{v}_j\}$ by the standard multivariate normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Start optimization from the factorization step (Eq. (3)).

Overall Performance w.r.t. Rank. As shown in Figure 2, the test performance improves as the rank of the latent intent space increases. But the improvement becomes less significant as the rank becomes larger since it starts to overfit the data.

Training RMSE. The training RMSE exhibit similar trends. They converge fast within a few iterations. All the models involving factorization achieve less than 0.1 RMSE. The pure regression achieves ~ 0.9 RMSE[5].

Test RMSE. We observe that the joint model with $\mathbf{w}$ updated first performs best. This is because the joint model takes into account the explicit features and latent intents simultaneously and balance them in the optimized way. The pure regression model also has good generalizability. The joint model with $\mathbf{w}$ updated last follows and the pure matrix factorization model performs worst. The difference between the two initialization settings indicates that a good starting point is crucial for the local optimum attained, which is not surprising. A sparse matrix completion problem can have many local optimal points. The pure regression model provides a good starting point with a reasonable $\mathbf{w}$ which leads the algorithm to converge to a better objective value. The pure regression model is better than the pure matrix factorization model. This aligns with the intuition that high-quality features are more reliable than unsupervised clustering.

[5]The training RMSE of the pure regression model is hidden in the upper three sub-figures to ensure a better scale of the other curves.

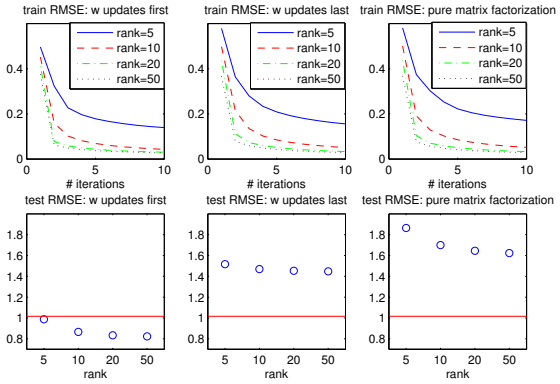


Figure 2: RMSE with Different Ranks of the Latent Intent Space with the Filtering Parameter 0.5.
At different ranks of the latent intent space, the upper three sub-figures plot the training RMSE as the iteration number goes from 1 to 10; the lower three sub-figures plot the test RMSE after 10 iterations. The red line in the lower three sub-figures represents the test RMSE of a pure regression on the features.

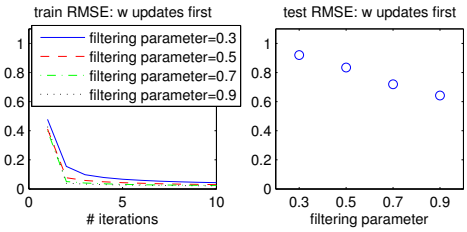


Figure 3: RMSE with Varying Filtering Parameter

RMSE with Varying Filtering Parameter

As the filtering parameter varies, the training and test RMSE at rank 20 for the joint model ($\mathbf{w}$ update first) is plotted in Figure 3. We observe that both training and test RMSE become smaller as the filtering parameter increases, which aligns with our intuition.

4.3 Extrinsic Evaluation - Impact on Search Ranking Relevance

Test Procedure. The evaluation is done with the standard machine learned ranking (MLR) system at Yahoo! on a set of editorial data, where the training data contains 548K (query,url,label) triples and the test data contains 86K triples. The labels were annotated by human editors. The MLR system learns a *ranking function* r based on the feature vectors $\{\mathbf{x}_{ql}\}$ for (query q, URL l) pairs in the training data. Originally, the ranking function is learned with $4,700$ features. We denote the original feature vector by $\mathbf{x}_{ql}^0$, which does not involve hostname level features. After we learn

the hostname preference, the preference score of the query q for the hostname of the URL l (denoted by $hostname(l)$) is fed into the MLR system as an additional feature and we learn a new ranking function. Instead of adding only the preference score feature, we add three new features to $\mathbf{x}_{ql}^0$ to reinforce the hostname preference. The three new features are respectively 1) the predicted hostname preference score: $\mathbf{w}^T\mathbf{f}_{q,hostname(l)} + \mathbf{u}_q^T\mathbf{v}_{hostname(l)}$; 2) the noisy hostname click number: $click(q, hostname(l))$; and 3) the explicit similarity score $\mathbf{u}_q'^T\mathbf{v}'_{hostname(l)}$.

The ranking results using the new ranking function are compared to the ranking results using the original ranking function on the test data by the measure of discounted cumulative gain (DCG). We examine the relative DCG gain under different parameter/model settings. The missing features due to insufficient coverage are all set to a system default value.

Performance w.r.t. the Relative DCG Gain. We sort the different model/parameter configurations by the relative DCG gain at position 5 and summarize the results in Table 1. We consider the combinations of the following three factors: filtering parameter $\theta \in \{0.3, 0.5, 0.7, 0.9\}$; the rank of the latent intent space $K \in \{20, 50\}$; and whether we use the joint model with $\mathbf{w}$ updated first or the pure matrix factorization model (pure MF).

Under various settings, we obtain a relative DCG gain of around 2%. This is impressive given that we are only adding 3 new features to the 4700 existing features. The highest DCG gain at position 1 we obtain is 2.80% and the number is 2.26% for position 5. In general, we also observe that the performance is better with a lower filtering parameter and using the joint model. In fact, a lower filtering parameter means better coverage of query-URL pairs. Too many missing values for the new features counteracts the benefit of a more accurate prediction. The rank 20 and 50 doesn't have significant difference, which indicates that it is not necessary to use a very high rank for the hostname preference learning due to potential overfitting.

4.4 Efficiency Study

We study the efficiency of our proposed algorithm on Hadoop, and compare with Spark implementations only[6]. The following algorithms are considered: 1) Blocked ALS for Pure Matrix Factorization in Spark MLlib. 2) The proposed block-wise parallel algorithm for feature aware matrix completion. 3) Plain block-wise implemention for feature aware matrix completion without fusing the weight update into the factorization step.

The training time with different filtering parameters at different ranks are shown in Figure 4. The algorithm is run on a commercial Hadoop cluster with normal amount of traffic. We use 20 worker nodes and 10 iterations. The maximum memory that can be used at each node is set to 4G. At rank 20, the proposed algorithm finishes within a few minutes. At rank 50, it finishes within 20 minutes. At rank 100, it finishes within 1 hour.

[6]As discussed in Section 2.3, traditional MapReduce on Hadoop is not suitable for iterative algorithms due to heavy overhead on disk IOs. Implementations with traditional MapReduce turn out to be slower than Spark implementations by several orders of magnitude.

Table 1: Relative DCG Gain. Sorted by DCG gain@5

θ	rank	model	DCG gain@1	DCG gain@5
0.3	50	joint	2.49%	2.26%
0.3	50	pure MF	2.64%	2.21%
0.3	20	pure MF	2.63%	2.19%
0.5	20	joint	2.64%	2.18%
0.3	20	joint	2.53%	2.17%
0.7	50	pure MF	2.80%	2.15%
0.5	50	joint	2.42%	2.14%
0.7	20	joint	2.66%	2.11%
0.7	20	pure MF	2.55%	2.10%
0.5	20	pure MF	2.53%	2.09%
0.9	50	joint	2.54%	2.06%
0.7	50	joint	2.50%	2.03%
0.5	50	pure MF	2.64%	2.02%
0.9	20	joint	2.50%	2.00%
0.9	20	pure MF	2.24%	1.94%
0.9	50	pure MF	2.36%	1.92%

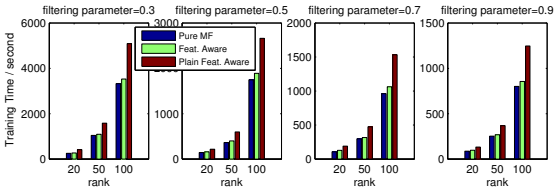


Figure 4: Training Time under Different Parameter Settings with 20 Worker Nodes and 10 Iterations

Compared to the pure matrix factorization, the proposed algorithm has very little overhead although it takes features into account. In fact, the most time consuming part is the communication among nodes, while computation within each node is fast in-memory process. As long as we distribute the data (the matrix to be completed and the features) at the very beginning to each worker node, there will be no additional communication cost in the later optimization stages. The plain implementation is slower than the proposed algorithm by almost half of the total time due to the same reason: Without fusing the weight update into the factorization step, the latent vectors $\{\mathbf{u}_i\}$ need to be sent to each corresponding vblocks one more time for the weight update, which induces additional communication time.

5 Conclusions

In this paper, we learn the hostname preference and investigate its impact on search ranking relevance. We observe impressive boost of search relevance by hostname preference modeling. The block-wise parallel solution for feature-aware matrix completion facilitates large scale experiments, which makes frequent model update and intensive comparative studies possible. Both the preference learning model and the optimization algorithm can go beyond the task in this paper. They apply to general preference learning applications where explicit features and latent intent are present simultaneously with scalability being critical.

Acknowledgments

We sincerely thank the anonymous reviewers for their useful comments. This work was done during the first author's internship at Yahoo! Labs. It was also sponsored in part by the U.S. Army Research Lab. under Cooperative Agreement No. W911NF-09-2-0053 (NSCTA), National Science Foundation IIS-1017362, IIS-1320617, and IIS-1354329, HDTRA1-10-1-0120, and grant 1U54GM114838 awarded by NIGMS through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (www.bd2k.nih.gov). The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies of the U.S. Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[Chandrasekar *et al.*, 2004] Raman Chandrasekar, Harr Chen, Simon Corston-Oliver, and Eric Brill. Subwebs for specialized search. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 480–481, New York, NY, USA, 2004. ACM.

[Diaz, 2009] Fernando Diaz. Integration of news content into web results. WSDM '09, pages 182–191, 2009.

[Gemulla *et al.*, 2011] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. KDD '11, 2011.

[Gibson *et al.*, 1998] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia : Links, Objects, Time and Space—structure in Hypermedia Systems: Links, Objects, Time and Space—structure in Hypermedia Systems*, HYPERTEXT '98, pages 225–234, New York, NY, USA, 1998. ACM.

[Hu *et al.*, 2009] Jian Hu, Gang Wang, Fred Lochovsky, Jian-tao Sun, and Zheng Chen. Understanding user's query intent with wikipedia. WWW '09, pages 471–480, 2009.

[Khanna *et al.*, 2013] Rajiv Khanna, Liang Zhang, Deepak Agarwal, and Bee-Chung Chen. Parallel matrix factorization for binary response. IEEE bigData'13, 2013.

[Kleinberg, 1999] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[König *et al.*, 2009] Arnd Christian König, Michael Gamon, and Qiang Wu. Click-through prediction for news queries. SIGIR '09, pages 347–354, 2009.

[Li *et al.*, 2008] Xiao Li, Ye-Yi Wang, and Alex Acero. Learning query intent from regularized click graphs. SIGIR '08, pages 339–346, 2008.

[Liu *et al.*, 2010] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. WWW '10, 2010.

[Ma *et al.*, 2011] Hao Ma, Chao Liu, Irwin King, and Michael R. Lyu. Probabilistic factor models for web site recommendation. SIGIR '11, 2011.

[Ren *et al.*, 2014] Xiang Ren, Yujing Wang, Xiao Yu, Jun Yan, Zheng Chen, and Jiawei Han. Heterogeneous graph-based intent learning with queries, web pages and wikipedia concepts. WSDM '14, pages 23–32, 2014.

[Shang *et al.*, 2014] Jingbo Shang, Tianqi Chen, Hang Li, Zhengdong Lu, and Yong Yu. A parallel and efficient algorithm for learning to match. ICDM'14, 2014.

[Song *et al.*, 2011] Yang Song, Nam Nguyen, Li-wei He, Scott Imig, and Robert Rounthwaite. Searchable web sites recommendation. WSDM '11, pages 405–414, 2011.

[Teflioudi *et al.*, 2012] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. ICDM '12, pages 655–664, 2012.

[Wu *et al.*, 2013] Wei Wu, Hang Li, and Jun Xu. Learning query and document similarities from click-through bipartite graph with metadata. WSDM '13, pages 687–696, 2013.

[Xue *et al.*, 2005] Gui-Rong Xue, Qiang Yang, Hua-Jun Zeng, Yong Yu, and Zheng Chen. Exploiting the hierarchical structure for link analysis. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 186–193, New York, NY, USA, 2005. ACM.

[Yu *et al.*, 2012] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. ICDM '12, 2012.

[Zaharia *et al.*, 2010] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud'10, 2010.

[Zhou *et al.*, 2008] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. AAIM '08, pages 337–348, 2008.