ISSN (Online) : 2319 - 8753
ISSN (Print) : 2347 - 6710

International Journal of Innovative Research in Science, Engineering and Technology

Volume 3, Special Issue 3, March 2014

2014 International Conference on Innovations in Engineering and Technology (ICIET'14)

On 21st&22ndMarch, Organized by

K.L.N. College of Engineering, Madurai, Tamil Nadu, India

Analyzing and Organizing User Search Histories Based On Query Logs

G.Manoranjitham, M.Ramesh

PG Student, Dept. of Computer Science and Engineering, Kalaignar Karunanidhi Institute of Technology, Coimbatore, India.

Assistant Professor Dept. of Computer Science and Engineering, Kalaignar Karunanidhi Institute of Technology, Coimbatore, India.

Abstract:Creating search histories is a difficult process in the web and organizing the user search logs is rapidly increasing in the field of data mining for finding the user interestingness. Daily billions of queries can be passed to the server for relevant information, most of the search engines retrieves the information based on the query similarity score or related links with respect to the given query. This paper explains the problem of organizing a user's historical queries into groups in a dynamic and automated fashion. This paper go beyond approaches that rely on textual similarity or time thresholds, and propose a more robust approach that leverages search query logs. The Incremental algorithm is used as IAssociation rule and ICover graph. This work experimentally study the performance of different techniques, and showcase their potential, especially when combined together.

Keywords - incremental algorithm, Iassociation rule, Icover graph, query logs

I.INTRODUCTION

A key challenge for web search engines is improving user satisfaction. Therefore, search engine companies exert significant effort to develop means that correctly "guess" what the real hidden intent behind a submitted query is. In the latest years, web search engines have started to provide users with query recommendations to help them refine queries and to quickly satisfy their needs. Query suggestions are generated according to a model built on the basis of the knowledge extracted from query logs. The model usually contains information on relationships between queries that are used to generate suggestions. Since the model is built on a previously collected snapshot of a query stream, its effectiveness decreases due to interest shifts. To reduce the effect of aging, query recommendation models must be periodically re-built or updated [3].

This paper proposes two novel incremental algorithms that update their model continuously on the basis of each new query processed. Designing an effective method to update a recommendation model poses interesting challenges due to:

i) *Limited memory availability* – queries are potentially infinite, and should keep in memory only those queries "really" useful for recommendation purposes,

ii) *Low response time* – recommendations and updates must be performed efficiently without degrading user experience.

The two proposed algorithms use two different approaches to generate recommendations. The first uses association rules for generating recommendations, and it is based on the *static* query suggestion algorithm, while the second uses click-through data[13].

The new class of query recommender algorithms proposed here "*incrementally updating*" query recommender systems to point out that this kind of systems update the model on which recommendations are drawn without the need for rebuilding it from scratch. There are multiple tests conducted on a large real-world query log to evaluate the effects of continuous model updates on the effectiveness and the efficiency of the query recommendation process. Result assessment used an evaluation methodology that measures the effectiveness of query recommendation algorithms by means of different metrics. Experiments show the superiority of incrementally updating algorithms with respect to their static counterparts. Moreover, the tests conducted demonstrated that our solution to update the model each time a new query is processed has a limited impact on system response time.

The main contributions presented in this work are: i) a novel class of query recommendation algorithms whose models are continuously updated as user queries are processed, ii) two new metrics to evaluate the quality of the recommendations computed, iii) an analysis of the effect of time on the quality and coverage of the suggestions provided by the algorithms presented and by their static counterparts.

II.RELATED WORK

In recent work, Jones and Klinkner [4] and Boldi et al. [5] investigate the search-task identification problem. More specifically, Jones and Klinkner [4] considered a search session to consist of a number of tasks (missions), and each task further consists of a number of subtasks (goals). They trained a binary classifier with features based on time, text, and query logs to determine whether two queries belong to the same task. Boldi et al. [5] employed similar features to construct a query flow graph, where two queries linked by an edge were likely to be part of the same search mission.

Our work differs from these prior works in the following aspects. First, the query-log based features in [4], [5] are extracted from co-occurrence statistics of query pairs. This paper additionally consider query pairs having common clicked URLs and this paper exploit both co-occurrence and click information through a combined query graph. Jones and Klinkner [4] will not be able to break ties when an incoming query is considered relevant to two existing query groups. Additionally, our approach does not involve learning and thus does not require manual labeling and retraining as more search data come in; our Markov random walk approach essentially requires maintaining an updated combined query graph. Finally, our goal is to provide users with useful query groups on-the-fly while respecting existing query groups. On the other hand, search task identification is mostly done at server side with goals such as personalization, query suggestions [5], etc.

III.SEARCH GOALS AND MISSION

Our goal is to automatically organize a user's search history into query groups, each containing one or more related queries and their corresponding clicks. Each query group corresponds to an atomic information need that may require a small number of queries and clicks related to the same search goal. For example, in the case of navigational queries, a query group may involve as few as one query and one click. For broader informational queries, a query group may involve a few queries and clicks.

DEFINITION 1. A query group is an ordered list of queries q_i, together with the corresponding set of clicked URLs, clk_i of q_i. A query group is denoted as $s = \langle\{q_1, clk_1\}, .., \{q_k, clk_k\}\rangle$.

A. *Dynamic Query Grouping*

To group query dynamically, this work first place the current query and clicks into a singleton query group $s_c = \{q_{c,} clk_c\}$, and then compare it with each existing query group s_i within a user's history. Determine if there are existing query groups sufficiently relevant to s_c. If so, merge s_c with the query group s having the highest similarity T_{max} above or equal to the threshold T_{sim}. Otherwise keep s_c as a new singleton query group and insert it into *S*.

B. *Calculation of Query Relevance*

To ensure that each query group contains closely related and relevant queries and clicks, it is important to have a suitable relevance measure *sim* between the current query singleton group s_c and an existing query group $s_i \in S$ [1]. There are a number of possible approaches to determine the relevance between current query and existing query.

$$sim_{time}(s_c, s_i) = \frac{1}{|time(q_c) - time(q_i)|}$$

$$sim_{jaccard}(s_c, s_i) = \frac{|words(q_c) \cap words(q_i)|}{|words(q_c) \cup words(q_i)|}$$

$$sim_{cor}(s_c, s_i) = \frac{|retrieved(q_c) \cap retrieved(q_i)|}{|retrieved(q_c) \cup retrieved(q_i)|}$$

$$sim_{ATSP}(s_c, s_i) = \frac{freq(q_c, q_i)}{freq(q_c)}$$

IV.QUERY RELEVANCE USING SEARCH LOGS

To calculate the query relevance based on web search logs, capture the two important properties of relevant queries[6]:

1) queries that frequently appear together as reformulations
2) queries that have induced the users to click on similar set of pages.

A. *Search Behavior*

The IAssociation Rule represents the relationship between a pair of queries that are likely reformulations of each other. The ICover Graph, represents the relationship between two queries that frequently lead to clicks on similar URLs. The query grouping merges the information from IAssociation rule and ICover graph. The above three methods are defined over the same set of vertices V_Q, consisting of queries which appear in at least one of the graphs, but their edges are defined differently.

V.INCREMENTAL ALGORITHM

Incremental algorithms are radically different from static methods for the way they build and use recommendation models. While static algorithms need an off-line pre-processing phase to build the model from scratch every time an update of the knowledge base is needed, incremental algorithms consist of a single online module integrating the two functionalities:

i) updating the model.

ii) providing suggestions for each query.

Starting from the two algorithms presented above, design two new query recommender methods continuously updating their models as queries are issued. Below algorithms formalize the structure of the two proposed incremental algorithms that are detailed in the following. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. Both algorithms exploit LRU caches and Hash tables to store and retrieve efficiently queries and links during the model update phase[13].

Our two incremental algorithms are inspired by the Data Stream Model in which streams of queries are processed by a database system. Queries consist modifications of values associated with a set of data. When the dataset fits completely in memory, satisfying queries is straightforward. Turns out that the entire set of data cannot be contained in memory. Therefore, an algorithm in the data stream model must decide, at each time step, which subset of the set of data is worthwhile to maintain in memory. The goal is to attain an approximation
of the results we would have had in the case of the non-streaming model. Make a first step towards a data stream model algorithmic framework aimed at building query recommendations.

The two algorithms considered use different approaches for generating recommendations. The first uses association rules while the second exploits click-through data.

Below Fig.1 explains entire work of this paper. User first enters the query for getting efficient results. The search engine compares the entered query with existing query log. If it is existed in the query log, the search engine applies incremental algorithm for that entry and provides results to user. The incremental algorithm includes IAssociation rule and ICover graph.

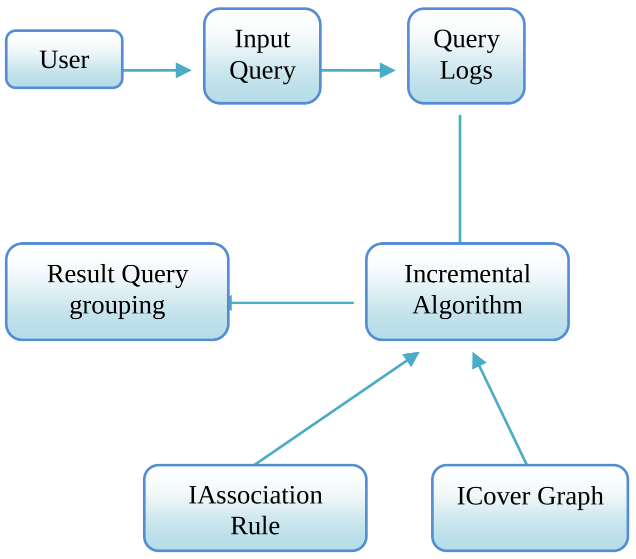

Figure 1. Architectural Diagram

A. *IAssociation Rules*

Algorithm1 specifies the operations performed by IAssociationRules, the incremental version of AssociationRules.

Algorithm 1. IAssociationRules

```
1: loop
2:     $(u, q) \leftarrow$ GetNextQuery() {Get the query q and the user u who submitted it}
3:     ComputeSuggestions $(q, \sigma)$ {Compute suggestions forquery q over σ}
4:     if ∃ LastQuery (u) then
5:         $q' \leftarrow$ LastQuery (u)
6:         LastQuery (u) $\leftarrow q$ {Update the last query submitted by u.}
7:         if $\exists\sigma_{q',q}$ then
8:             $++\sigma_{q',q}$ {Increment Support for $q' \Longrightarrow q$}
9:         else
10:            LRUInsert $(\sigma, (q', q))$ {Insert an entry for (q', q) in σ. If σ is full, remove an entry according to an LRU policy.}
11:        end if
12:    else
13:        LRUInsert (u, q, LastQuery) {Insert an entry for (u, q) in LastQuery. If LastQuery is full, remove an entry according to an LRU policy.}
14:    end if
15: end loop
```

The data structures storing the model are updated at each iteration. This work uses the LastQuery auxiliary data structure to record the last query submitted by *u*.

Since the model and the size of LastQuery could grow indefinitely, whenever they are full, the LRUInsert function is performed to keep in both structures only the most recently used entries.

Claim. Keeping up-to-date the AssociationRule-based model is *O* (1).

The proof of the claim is straightforward. The loop at line 3 of Algorithm 1 is made up of constant-cost operations. LRUInsert has been introduced to maintain the most recently submitted queries in the model[7].

B. ICover Graph

The incremental version of CoverGraph adopts a solution similar to that used by IAssociationRules. It uses a combination of LRU structures and associative arrays to incrementally update the (LRU managed) structure σ. Algorithm 2 shows the description of the algorithm.The hash table queryHasAClickOn is used to retrieve the list of queries having c among their clicked URLs. This data structure is stored in a fixed amount of memory, and whenever its size exceeds the allocated capacity, an entry is removed on the basis of a LRU policy (this justifies the conditional statement at line 6)[13].

Claim. Keeping up-to-dated a CoverGraph-based model is *O*(1).

Actually, the cost depends on the degree of each query/node in the cover graph.

i. the degree of nodes in the cover graph follows a power-law distribution
ii. the maximal number of URLs between two queries/nodes is constant, on average. The number of iterations needed in the loop at line 11 can be thus considered constant.

From the above methods, it is clear that to effectively produce recommendations; a continuous updating algorithm should have the following characteristics:

- The algorithm must cope with an undefined number of queries. LRU caches can be used to allow the algorithm to effectively keep in memory only the most relevant items for which it is important to produce recommendations.
- The lookup structures used to generate suggestions and maintain the models must be efficient, possibly constant in time. Random-walks on graph-based structures, or distance functions based on comparing portions of texts, etc., are not suitable for our purpose.
- A modification of an item in the model must not involve a modification of the entire model. Otherwise, update operations take too much time and jeopardize the efficiency of the method.

Algorithm 2. ICoverGraph

```
1: Input: A threshold $\tau$ .
2: loop
3:     $(u, q)$ ← GetNextQuery() {Get the query q and the user u who submitted it.}
4:     ComputeSuggestions $(q, \sigma)$ {Compute suggestions for query q over σ.}
5:     c = GetClicks $(u, q)$
6:     if ∃ queryHasAClickOn(c) then
7:         queryHasAClickOn (c) ← q
8:     else
9:         LRUInsert (queryHasAClickOn, c)
10:    end if
11:    for all $q' \neq q \in$ queryHasAClickOn(c) s.t. $W((q, q)) > \tau$ do
12:        if $w > \tau$ then
13:            if $\exists \sigma_{q,q'}$ then
14:                $\sigma_{q,q'} = w$
15:            else
16:                LRUInsert $(\sigma, (q', q), w)$
17:            end if
18:        end if
19:    end for
20: end loop
```

VI.QUERY RELEVANCE CALCULATION

For a given query *q*, compute a relevance vector, where each query corresponds to the relevance value of each query $q_j \in V_Q$ to q. The edges in result query graph correspond to pairs of relevant queries extracted from the query logs and the click logs.

Let us consider a vector r_q, where each entry, $r_q(q_j)$, is $w_f(q, q_j)$ if there exists an edge from q to q_j in and 0 otherwise. One straightforward approach for computing the relevance of q_j to q is to use this $r_q(q_j)$ value. However, although this may work well in some cases, it will fail to capture relevant queries that are not directly connected in result query graph (and thus $r_q(q_j) = 0$).

Therefore, for a given query *q*, suggest a more generic approach of obtaining query relevance by defining a Markov chain for *q*, MC_q, over the given graph, result query graph, and computing the stationary distribution of the chain. This paper refer to this stationary distribution as the fusion relevance vector of *q*, rel_q^F, and use it as a measure of query relevance.

The stationary probability distribution of MC_q can be estimated using the matrix multiplication method, where the matrix corresponding to MC_q is multiplied by itself iteratively until the resulting matrix reaches a fix point. However, given our setting of having thousands of users issuing queries and clicks in real time and the huge size of result query group, it is infeasible to perform the expensive matrix multiplication to compute the stationary distribution whenever a new query comes in. Instead, pick the most efficient Monte Carlo random walk simulation method among the ones presented in [15], and use it on result query group to approximate the stationary distribution for *q*.

Algorithm 3. Query Relevance Calculation
Relevance(q)

```
Input
  1: the result query group (combined)
  2: the jump vector, $g$
  3: the damping factor, $d$
  4: the total number of random walks, numRWs
  5: the size of neighbourhood, maxHops
  6: the given query, $q$
Output: the fusion relevance vector for $q$, $rel_q^F$
  1: Initialize $rel_q^F = 0$
  2: num Walks=0; numVisits=0
  3: While numWalks < numRWs
  4:     numHops=0; v=q
  5:     while $v \neq NULL \wedge numHops < maxHops$
  6:         numHops++
  7:         $rel_q^F(v)$++; numVisits++
  8:         v=SelectNextNodeToVisit(v)
  9:     numWalks++
  10: For each $v$, normalize $rel_q^F(v) = rel_q^F(v)/numVisits$
```

The algorithm 3. computes the fusion relevance vector of a given query q, rel_q^F. It requires the following inputs in addition to result query group. First, introduce a jump vector of q, g_p, that specifies the probability that a query is selected as a starting point of a random walk. Since set $g_p(q')$ to 1 if q'=q, and 0 otherwise, q will always be selected; the next section will generalize g_p to have multiple starting points by considering both q and the clicks for q. A damping factor, $d \in [0,1]$ (similar to the original Page Rank algorithm [16]), determines the probability of random walk restart at each node.

Two additional inputs control the accuracy and the time budget of the random walk simulation: the total number of random walks, *numRWs*, and the size of neighborhood explored, *maxHops*. As *numRWs* increases, the approximation accuracy of the fusion relevance vector improves by the law of large numbers. This work limit the length of each random walk to *maxHops*, assuming that a transition from q to q' is very unlikely if no user in the search logs followed q by q' in less than *maxHops* number of intermediate queries. In practice, we typically use *numRWs=1*,000,000 and *maxHops* =5, but reduce the number of random walk samples or the lengths of random walks by decreasing both parameters for a faster computation of rel_q^F [10].

The random walk simulation then proceeds as follows: use the jump vector g_p to pick the random walk starting point. At each node v, for a given damping factor d, the random walk either continues by following one of the outgoing edges of v with a probability of d, or stops and restarts at one of the starting points in g_p with a probability of *(1-d)*. Then, each outgoing edge, (v, q_i), is selected with probability $w_f(v, q_i)$, and the random walk always restarts if v has no outgoing edge. The selection of the next node to visit based on the outgoing edges of the current node v in result query graph and the damping factor d is performed by the *SelectNextNodeToVisit* process in Step (8) of the algorithm, which is illustrated in Algorithm 4. Notice that each random walk simulation is independent of another, so can be parallelized.

Algorithm 4. SelectNextNodeToVist(v)

```
Input:
  1: the query fusion graph, combined query
  2: the jump vector, $g$
  3: the damping factor, $d$
  4: the current node, $v$
Output:
  1: if random( ) < d
  2:     $V = \{q_i | (v, q_i) \in \varepsilon_{QF}\}$
  3:     pick a node $q_i \in V$ with probability
                $w_f(v, q_i)$
  4: else
  5:     $V = \{q_i | g(q_i) > 0\}$
  6:     pick a node $q_i \in V$ with probability $g(q_i)$
  7: return $q_i$
```

After simulating *numRWs* random walks on the result query group starting from the node corresponding to the given query q, normalize the number of visits of each node by the number of all the visits, finally obtaining rel_q^F, the fusion relevance vector of q. Each entry of the vector, $rel_q^F(q')$, corresponds to the fusion relevance score of a query $q' \in V_Q$ to the given query q. It is the probability that q' node is visited along a random walk originated from q node over the result query group.

Lastly, this work show that there exists a unique fusion relevance vector of a given query q, rel_q^F. It is well known that for a finite ergodic Markov chain, there exists a unique stationary distribution. In fact, the random walk simulation algorithm described in algorithm 3 approximates rel_q^F that corresponds to the stationary distribution of the Markov chain for q, MC_q [2].

VII.QUERY GROUPING

For each query, maintain a query image, which represents the relevance of other queries to this query. For each query group, maintain a context vector, which aggregates the images of its member queries to form an overall representation. This paper then propose a similarity function sim_{rel} for two query groups based on these concepts of context vectors and query images.

Context Vector: For each query group, maintain a context vector which is used to compute the similarity between the query group and the user's latest singleton query group. The context vector for a query group s, denoted cxt_s, contains the relevance scores of each query in V_Q to the query group s, and is obtained by aggregating the fusion relevance vectors of the queries and clicks in s. If s is a singleton query group containing only $\{q_{s1}, clk_{s1}\}$, it is defined as the fusion relevance vector $rel_{(q_{s1}, clk_{s1})}$. For a query group $s = \langle\{q_{s1}, clk_{s1}\}, \ldots, \{q_{sk}, clk_{sk}\}\rangle$ with $k > 1$, there are a number of different ways to define cxt_s. For instance, define it as the fusion relevance vector of the most recently added query and clicks, $rel_{(q_{s1}, clk_{s1})}$. Other possibilities include the average or the weighted sum of all the fusion relevance vectors of the queries and clicks in the query group.

Query Image: The image of q, denoted $\boldsymbol{I}(q)$ that expresses q as the set of queries in V_Q that are considered highly relevant to q. Generate $\boldsymbol{I}(q)$ by including every query q' whose relevance value to q, $rel_q(q')$, is within top-X percentage. To do this, sort the queries by relevance, and find the cutoff such that the sum of the relevance values of the most relevant queries accounts for X% of the total probability mass.

Online Query Grouping. The similarity metric that operates on the images of a query and a query group. Some applications such as query suggestion may be facilitated by fast on-the-fly grouping of user queries. For such applications, avoid performing the random walk computation of fusion relevance vector for every new query in real time, and instead precompute and cache these vectors for some queries in our graph. This works especially well for the popular queries. In this case, essentially trading-off disk storage for runtime performance. Estimate that to cache the fusion relevance vectors of 100 million queries, require disk storage space in the hundreds of gigabytes. This additional storage space is insignificant relative to the overall storage requirement of a search engine. Meanwhile, retrieval of fusion relevance vectors from the cache can be done in milliseconds.

VIII.EXPERIMENTS

A. Experiments

This section provides study of behavior and performance of our algorithms on partitioning a user's query history into one or more groups of related queries. This work conducted experiments on a collection consisting of the first 3,200,000 queries from the AOL query log [14]. The AOL data-set contains about 20 million queries issued by about 650,000 different users, submitted to the AOL search engine.

B. Results

Result obtained graphs of IAssociation rule and ICover graph by merging a number of monthly search logs from a commercial search engine. Each monthly snapshot of the query log adds approximately 24 percent new nodes and edges in the graph compared to the exactly preceding monthly snapshot, while approximately 92 percent of the mass of the graph is obtained by merging nine monthly snapshots.

To reduce the effect of noise and outliers, pruned the IAssociation rule graph by keeping only query pairs that appeared at least two times and the ICover graph by keeping only queryclick edges that had at least 10 clicks. Based on these two graphs, constructed the combined graph. In order to create test cases for our algorithms, used the search activity (comprising at least two queries) of a set of 200 users (henceforth called the Rand200 data set) from our search log. To generate this set, users were picked randomly from our logs, and two human labelers examined their queries and assigned them to either an existing group or a new group if the labelers deemed that no related group was present.

A user's queries were included in the Rand200 data set if both labelers were in agreement in order to reduce bias and subjectivity while grouping. The labelers were allowed access to the web in order to determine if two seemingly distant queries were actually related. The average number of groups in the data set was 3.84 with 30 percent of the users having queries grouped in more than three groups. To measure the quality of the output groupings, for each user, start by computing query pairs in the labeled and output groupings. Two queries form a pair if they belong to the same group, with lone queries pairing with a special "null" query.

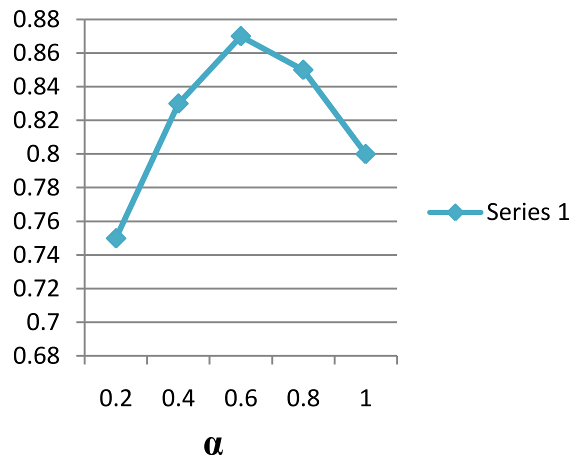

Figure 2: varying mix of query and click graph

The result is shown in Fig. 2; the horizontal axis represents α (i.e., how much weight we give to the query edges coming from the query reformulation graph), while the vertical axis shows the performance of our algorithm. From the graph, our algorithm performs best when α is around 0.7, with the two extremes (only edges from clicks, i.e., $\alpha = 0.0$ or only edges from reformulations, i.e., $\alpha = 1.0$) performing lower. It is interesting to note that, based on the shape of the graph, edges coming from query reformulations are deemed to be slightly more helpful compared to edges from clicks. This is because there are 17 percent fewer click-based edges than reformulation-based edges, which means that random walks performed on the query reformulation graph can identify richer query images.

IX.CONCLUSION

This paper studied the effects of incremental model updates on the effectiveness of two query suggestion algorithms. As the interests of search-engine users change over time and new topics become popular, the knowledge extracted from historical usage data can suffer an aging effect. Consequently, the models used for recommendations may rapidly become unable to generate high-quality and interesting suggestions. This work introduced a new class of query recommender algorithms that update "incrementally" the model on which recommendations are drawn. Starting from two state-of-the-art algorithms, designed two new query recommender systems that continuously update their models as queries are issued. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. In addition, proposed an automatic evaluation mechanism based on two new metrics to assess the effectiveness of query recommendation algorithms.

REFERENCES

[1] H.Hwang, W.Lauw, Lise Getoor and A.Ntowas, *Organizing User Search Histories*, IEEE Transaction Vol.24, No.5, May 2012.

[2] J. Teevan, E. Adar, R. Jones, and M.A.S. Potts, "Information Re-Retrieval: Repeat Queries in Yahoo's Logs," Proc. 30th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '07), pp. 151-158, 2007.

[3] A. Broder, "A Taxonomy of Web Search," SIGIR Forum, vol. 36, no. 2, pp. 3-10, 2002.

[4] A. Spink, M. Park, B.J. Jansen, and J. Pedersen, "Multitasking during Web Search Sessions," Information Processing and Management, vol. 42, no. 1, pp. 264-275, 2006.

[5] R. Jones and K.L. Klinkner, "Beyond the Session Timeout: Automatic Hierarchical Segmentation of Search Topics in Query Logs," Proc. 17th ACM Conf. Information and Knowledge Management (CIKM), 2008.

[6] P. Boldi, F. Bonchi, C. Castillo, D. Donato, A. Gionis, and S. Vigna, "The Query-Flow Graph: Model and Applications," Proc. 17th ACM Conf. Information and Knowledge Management (CIKM), 2008.

[7] D. Beeferman and A. Berger, "Agglomerative Clustering of a Search Engine Query Log," Proc. Sixth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD), 2000.

[8] R. Baeza-Yates and A. Tiberi, "Extracting Semantic Relations from Query Logs," Proc. 13th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD), 2007.

[9] J. Han and M. Kamber, Data Mining: Concepts and Techniques. Morgan Kaufmann, 2000.

[10] W. Barbakh and C. Fyfe, "Online Clustering Algorithms," Int'l J. Neural Systems, vol. 18, no. 3, pp. 185-194, 2008.

[11] Lecture Notes in Data Mining, M. Berry, and M. Browne, eds. World Scientific Publishing Company, 2006.

[12] V.I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," Soviet Physics Doklady, vol. 10, pp. 707- 710, 1966.

[13] D.Broccolo, O.Frieder, F.Maria Nardini, R.Perego and F.Silvestri, "Incremental Algorithms for Effective and Efficient Query Recommendations" Springer Trans SPIRE 2010.

[14] J.-R. Wen, J.-Y. Nie, and H.-J. Zhang, "Query Clustering Using User Logs," ACM Trans. in Information Systems, vol. 20, no. 1, pp. 59-81, 2002.

[15] A. Fuxman, P. Tsaparas, K. Achan, and R. Agrawal, "Using the Wisdom of the Crowds for Keyword Generation," Proc. the 17th Int'l Conf. World Wide Web (WWW '08), 2008.