Effective Query Recommendation with Medoid-based Clustering using a Combination of Query, Click and Result Features

Elham Esmaeeli-Gohari
Faculty of Computer Engineering, Yazd University, Yazd, Iran
elhamesmaeeli@stu.yazd.ac.ir
Sajjad Zarifzadeh*
Faculty of Computer Engineering, Yazd University, Yazd, Iran
szarifzadeh@yazd.ac.ir

Received: 20/Nov/2019 Revised: 21/Dec/2019 Accepted: 29/Feb/2020

Abstract
Query recommendation is now an inseparable part of web search engines. The goal of query recommendation is to help users find their intended information by suggesting similar queries that better reflect their information needs. The existing approaches often consider the similarity between queries from one aspect (e.g., similarity with respect to query text or search result) and do not take into account different lexical, syntactic and semantic templates exist in relevant queries. In this paper, we propose a novel query recommendation method that uses a comprehensive set of features to find similar queries. We combine query text and search result features with bipartite graph modeling of user clicks to measure the similarity between queries. Our method is composed of two separate offline (training) and online (test) phases. In the offline phase, it employs an efficient *k*-medoids algorithm to cluster queries with a tolerable processing and memory overhead. In the online phase, we devise a randomized nearest neighbor algorithm for identifying most similar queries with a low response-time. Our evaluation results on two separate datasets from AOL and Parsijoo search engines show the superiority of the proposed method in improving the precision of query recommendation, e.g., by more than 20% in terms of p@10, compared with some well-known algorithms.

Keywords: Recommendation Systems; Search Engine; Clustering; Query; Click.

1- Introduction

Nowadays, search engines play a critical role in giving users access to their needed information over the Internet. The user starts the search process via submitting a query to the search engine. The search engine processes the query and returns a result page containing an ordered list of URLs. The user reviews the excerpt of each URL and chooses some of them for further examination. If the user is satisfied with the returned information, the search process will be finished; otherwise, he/she submits another query. This process may include several cycles from submitting query to browsing results to be finished [1].
One of the most important challenges of search engines is to reduce the response time of queries, while returning the most relevant results. Both issues have a straight impact on users' satisfaction and also the efficiency of search engines. Most of the time, it is hard for users to precisely and clearly express their needs through submitted queries [2]. Therefore, search engines should smartly recommend queries that could lead users faster to the information they want, while keeping the semantic relevant to the original, submitted query [3, 4, 5]. In such cases, it is helpful to use the historical data collected from interactions of users with the search engine (e.g., past queries and clicks) [6].
There are several factors that make the problem of identifying users' intention from search challenging for information retrieval systems. One factor is the existence of polysemous words in queries [7]. For example, the term "jaguar" could refer either to a feline from the same genus as cheetahs, or to the vehicle brand of Jaguar car. There are hundreds of similar examples in other languages as well. For example, the term "شیر" in Persian means milk, lion, and faucet. On the other hand, there are many words that are synonym (e.g., crash, accident, and collision) and different users may use different queries for the exactly same topic.
In the past 15 years, three major approaches have been proposed for query recommendation. In the most popular approach, queries are clustered based on their similarity (e.g., with respect to content or result) and then most similar queries are recommended from closest clusters. There is also another approach that constructs a profile for each user via storing his/her past interactions with the search engine and then recommends queries according to the constructed profile. The final approach employs graph modeling in which a graph structure is built between

* Corresponding Author

queries using either subsequent queries in a session or common URL clicks made after queries. This graph is then used to find most similar queries.

The current approaches usually define the notion of similarity between queries from one aspect (e.g., text, result or click similarity), while neglecting other important features shared between similar queries. This can negatively affect the precision of query recommendation. In addition, the high processing and memory overhead of running recommendation algorithms is prohibitive in many cases. The aim of this research is to overcome the shortcoming of existing algorithms while taking their advantages, such that a high precision is attained while preserving the system efficiency in terms of CPU and RAM usages.

In this paper, we propose a new query recommendation method that combines the benefits of two well-known approaches, i.e., query clustering and graph modeling, together. Our method is composed of two separate offline and online phases. In the offline phase, we cluster our training queries based on three kinds of features, i.e., query n-grams, bipartite query-click graph and search results. The combination of these features can accurately identify relevant queries. Then, we use an efficient version of *k*-medoids algorithm to cluster queries. In the online phase, a fast randomized algorithm is proposed to approximate the most similar queries from the closest cluster. Our evaluations on the search log of two real search engines, Parsijoo (the first Persian web search engine) and AOL [8], show that the proposed method can achieve at least 7% better precision and 23% higher p@10 in comparison with three well-known query recommendation algorithms. Moreover, it incurs a low response time with an affordable memory overhead.

The rest of the paper is organized as follows: In the next section, we review the related works. Section 3 is devoted to the explanation of the proposed method. The results of our evaluations are presented in Section 4. Finally, we conclude the paper in Section 5.

2- Related Works

Query recommendation can be done in different forms, from spelling correction to query auto-completion and query expansion. As an early attempt, Zhang et al. propose to model users' sequential search behavior for recommending queries [9]. Since then, different techniques have been suggested for query recommendation which can be generally divided into query clustering [10, 11, 12, 13], constructing user profile [14, 15, 16] and graph modeling [17-22] categories.

In clustering-based techniques, queries are clustered and then recommended according to their similarity with respect to various factors. For example, Baeza-Yates et al., represent each query with a term-weight vector [11]. Their idea is that queries that have the same meaning may not have common terms, but they may have similar terms in documents selected by users. The term-weight vector of each query is calculated by aggregating the term-weight vectors of documents clicked as a result of query. Each term is weighted based on the number of its occurrences in the clicked documents as well as the number of clicks on documents containing that term. After that, clustering methods are used to find out similar queries. Chaudhary et al., propose to cluster queries based on query content and also the history of users' clicks [13]. In their paper, two principles are used: 1) if two submitted queries contain the same or similar terms, they probably convey the same information needs and therefore they can be located in the same cluster; 2) if two submitted queries lead to clicks on the same URLs, they are similar.

Some other works on query recommendation construct a profile for each user which includes all important interactions of user with the search engine (e.g., submitted queries and clicked URLs). The main idea is to use the users' search history to better understand their intention of search and resolve ambiguities for recommending queries [15]. In early works, there were three ways to construct user profiles: (1) utilizing user relevance feedback: the user is asked to specify whether the visited documents are relevant to his/her needs or not. If the document is relevant, it will be used to identify user's real needs [14]; (2) utilizing user preference: at the time of signing up, the user is asked to submit his preferences and personal information, such as interests, age and education background; (3) utilizing user ranking: the user is asked to rank each of the specified documents from 5 (very bad) to 1 (very good), based on its relevance to his information needs.

The above methods are costly from the viewpoint of users; they often prefer easier and faster methods. In [15], the user profile is constructed using the frequency of each term in user's visited documents. In other words, a user profile consists of a set of vectors where each vector represents a user session in which keywords of the documents clicked during the session along with their frequencies in these documents are stored. The aggregation of these vectors is then used for query recommendation. Since the current session denotes users' recent preferences, the weight of current session is set to a larger value than past sessions. Recently, in [16], the authors propose to construct user profiles for re-ranking search results. They first classify user's clicked URLs into hierarchical categories and then constructs user's profile with respect to these categories.

The third approach for query recommendation is based on graph modeling. Zhang et al. introduce a graph structure where vertices indicate queries and edges between vertices represent the textual similarity between corresponding queries [9]. An edge is weighted by a damping factor, which denotes the similarity of two consecutive queries in the same session. The similarity of two queries that are not neighbors is obtained by multiplying the weight of edges over the path joining them. In [17], the "Query-URL" bipartite graph is used for query recommendation. In this graph, vertices are queries and URLs, and each edge $e = (q_i, u_j)$ exists in the graph, if and only if, URL u_j is clicked by a user as a result of query q_i. The weight of edge e denotes the number of times users have access to URL u_j via query q_i. After constructing the graph, the similarity between queries is computed using a random walk technique.

Another graph modeling method is to use query-flow graph within sessions [18]. The idea is to utilize the sequence of queries and the modifications that users make to their queries until they reach their intended results. The query-flow graph is constructed based on queries that users submit in different sessions. Here, vertices are distinct queries and each edge (q_i, q_j) indicates that at least in one session query q_j has been submitted right after query q_i. The weight of edge (q_i, q_j) is assigned based on two factors: 1) the probability that two queries q_i and q_j belong to the same search mission and 2) the relative frequency of (q_i, q_j) pair and query q_i. The drawback of this method is that almost half of query pairs occur only once in users' searches, and therefore this graph is sparse. On the other hand, the query-flow graph is asymmetric, because more than 90% of the edges are not reciprocal. Hence, it is not certain that two neighbor queries are equivalent [18].

In order to improve the query-flow graph, a framework is proposed in [19], in which users' querying behaviors are modeled as a Markov-chain process. Furthermore, Bai et al. propose a new approach using an intent-biased random walk algorithm to reduce the sparsity problem in query-flow graphs [20]. The main disadvantage of two latter works is that their accuracy is fairly low, since they do not use click information during graph construction.

Some recent works have adopted different approaches for query recommendation. In [23], the authors propose a context-aware query recommendation which uses a sequence of sequence models (seq2seq) along with a version of Hierarchical Neural Networks for encoding submitted queries in a session and producing the best possible sequence of terms as the next query. This approach considers each query merely as a sequence of terms and hence, it takes into account neither click nor result level information.

In [24], a Knowledge Base (KB) technique is introduced to generate query recommendations based on named entities existing in queries. In order to improve the precision for short queries or queries that their entities are not identifiable, two hybrid methods (named as KB-QREC and D-QREC) are suggested in [25] where KBs and click information are used to retrieve entity relationship information.

To devise a practical recommendation method, we should make a trade-off between the precision and the computational performance of recommendation algorithm. In this paper, we suggest to use a complete set of features along with efficient clustering and nearest neighbor algorithms to improve both parameters at the same time.

3- The Proposed Method

The general workflow of our method is illustrated in Fig. 1. Our method comprises two separate offline and online phases (or equivalently, train and test phases). In the offline phase, we preprocess our training data, obtained from data logs of search engines, via performing spell checking, writing unification, stemming, synonym labeling and stop word removal tasks. Then, we cluster training queries using three kinds of extracted features: "N-grams feature", "Bipartite graph feature" and "Top-k search results and their ranks feature". We employ a simple and fast k-medoids algorithm for query clustering. In the online phase, upon the arrival of a test query, we first preprocess the query (just as the offline phase) and then find similar queries through identifying the closest cluster to that query and finally approximating the most similar queries inside that cluster.

3-1- Data Preprocessing Step

In the preprocessing step (which is common in both offline and online phases), the writing structure of queries is unified as much as possible. This step involves the following tasks:

Spell checking: the spelling errors in queries are corrected. For example, the query "TOFL exam" is modified to "TOEFL exam" or the Persian query "Tarneh Alidoosi" [as an Iranian actress] is modified to "Taraneh Alidoosti".

Writing unification: the punctuation marks (? : ! . - ,) are removed, since these marks result in different forms of writing a word. On the other hand, some of the letters in Persian and Arabic alphabets have different shapes, but are used interchangeably (for example, ی and ئ). Therefore, such letters are normalized and replaced with the same Unicode. Furthermore, plural signs are removed. For example, the query "programs" is modified to "program". Also, all English letters are converted to lower case. For instance, both queries "JAVA" and "Java" are converted to "java".

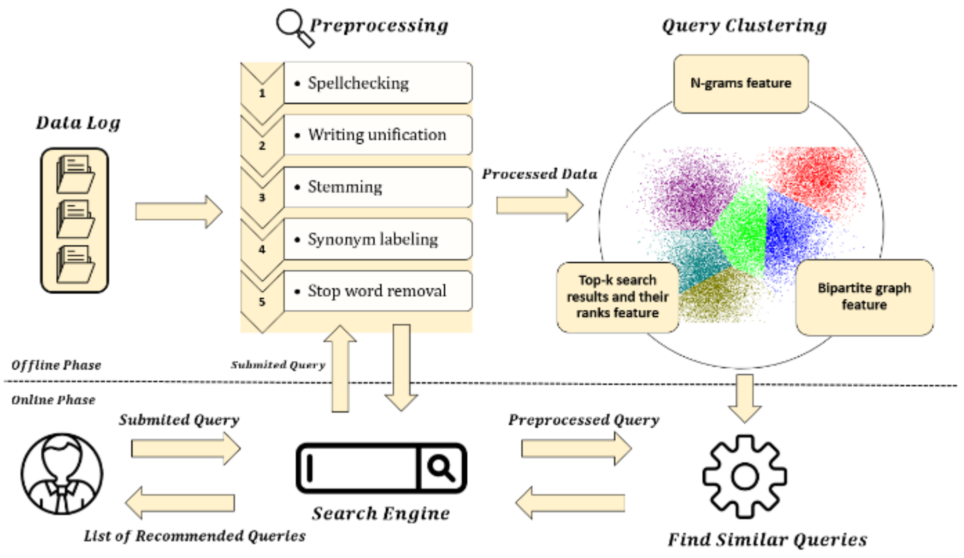

Fig. 1: Workflow of the proposed method

Stemming: All words are replaced with their stems. In this way, the diversity of words in queries is reduced which increases the chance of detecting similar queries. For example, if a user submits a query containing the word “analyze”, then queries containing the words “analysis”, “analyzing”, “analyzer”, “analyzes” and “analyzed” could also be related to the user’s search needs. Previous studies have shown that stemming reduces the size of bag-of-words by about 50% [26].

Synonym labeling: All words with the same meaning are labeled with the same label such that the distance between similar queries are reduced. As a result of this task, each word is replaced with its most frequently used synonym. For example, the words “battle”, “clash”, “attack” and “fight” are synonyms and therefore the word “fight” is used as their label.

Stop word removal: The stop words are identified and removed from the submitted queries by using a fixed list of stop words. The stop words are words that appear frequently in queries and documents, but do not imply any particular meaning. Such words consist of articles, prepositions, conjunctions, pronouns and some specific nouns and verbs. As an example, terms like “however” are removed from queries. This task improves both the precision and the computational performance of recommendation system.

These tasks are all performed, using open-source tools available for Persian and English languages, i.e., Hunspell package [27] and PersianStemmer [28].

3-2- Query Clustering Step

Due to its simplicity and performance gains, query clustering has attracted the attention of many researchers in the past years [12, 13]. After accomplishing the data preprocessing step, we cluster our training queries. We argued earlier that terms cannot solely represent the purpose of a query and as a result, it is not precise to just consider the terms of queries to measure their similarity. In this paper, we use various lexical and semantic features to find out similar queries. After the data preprocessing phase, training queries are clustered. In order to compute the similarity between queries, the following three features are extracted:

- N-grams feature
- Bipartite graph (bigraph) feature
- Top-k search results feature

3-2-1- N-grams Feature

If two queries have the same words, they are supposed to indicate similar information needs. To obtain the lexical similarity between queries, we extract the N-grams (up to trigram) of each query. For example, for the sample query “find research council site”, we have:

- Unigrams: “find”, “research”, “council”, “site”;
- Bigrams: “find research”, “research council”, “council site”;
- Trigrams: “find research council”, “research council site”.

We construct an N-gram vector for each query. Our vocabulary is the set of all different 1/2/3-grams in submitted queries. An N-gram g exists in the vector of query q, if g appears in q at least once. Similar to the previous works such as [11], the N-gram vector is then

used to obtain the similarity measure between queries. For instance, the vector for the above example includes the following N-grams:
{find, research, council, sites, find research, research council, council sites, find research council, research council sites}
We use the simple Jaccard similarity [35] to compute the similarity between two queries q_x and q_y:

$$Sim_{ngrams}(q_x, q_y) = \frac{|Ngrams(q_x) \cap Ngrams(q_y)|}{|Ngrams(q_x) \cup Ngrams(q_y)|} \quad (1)$$

where $Ngrams(q)$ represents the set of N-grams exist in the vector of query q. In other words, the above similarity indicates the ratio of common N-grams between two queries. The average number of terms in queries is usually limited to a small value, e.g., around 4 in our dataset. Thus, it is easy to compute the above formula. The obtained similarity from the N-grams feature would be more reliable for long queries and may fail for short queries. For example, consider two queries “Apple pear” as a fruit, and “Apple Inc.” as a company. According to this feature, the similarity between two queries is 20%, despite the fact that they convey different meanings. On the other hand, many queries (like “natural disasters” and “unforeseen events”) denote the same purpose, but they are expressed with different terms. We need extra features to unravel such problems.

3-2-2- Bipartite Graph (Bigraph) Feature

The second feature is based on this assumption that queries that lead to a click on the same documents are likely to represent the same information needs [13]. For this reason, the “Query-URL” bipartite graph (bigraph) is used to obtain the second feature. In this graph, vertices are queries and ULRs. Each edge (q, u) in the graph shows that at least one user has clicked on URL u, as the search result of query q. The weight of the edge shows the number of users’ clicks on URL u after query q. All other queries that lead to click on u are nominated for recommendation for query q. We use $G(Q + U, E)$ to refer to the bipartite graph where Q and U are the set of all queries and URLs, respectively which aggregately construct the set of vertices and E is the set of edges in the graph. We use $w[q, u]$ to denote the weight of the edge between query q and URL u.
From the bipartite graph, we construct a URL vector for each query in which a URL u exists in the vector of query q (with a weight of $w[q, u]$), if (q, u) exists in the graph G. Just as the previous works [9, 17], in order to compute the similarity between two queries q_x and q_y from the bipartite graph, the following steps should be taken:

- The set of URLs that appear in the URL vectors of both q_x and q_y are extracted. We use CU_{xy} to refer to this set.
- If $CU_{xy} = \emptyset$, the similarity between q_x and q_y is assumed zero.
- Otherwise, the similarity between q_x and q_y is obtained using the following equation:

$$Sim_{bigraph}(q_x, q_y) = \frac{\sum_{i=1}^{|CU_{xy}|}(w[q_x, CU_{xy}(i)] + w[q_y, CU_{xy}(i)])}{\sum_{i=1}^{|U|}(w[q_x, u_i] + w[q_y, u_i])} \quad (2)$$

where $CU_{xy}(i)$ is the i^{th} URL in set CU_{xy} and u_i is the i^{th} URL in set U. Moreover, $w[q_x, CU_{xy}(i)]$ denotes the weight of the edge between query q_x and URL $CU_{xy}(i)$ in the bigraph.
Fig. 2 illustrates a sample “Query-URL” bipartite graph. The number over each link shows the weight of that link. Suppose we want to obtain the similarity between q_5 and other queries in this graph. As shown in the figure, users have clicked on URLs u_1, u_2 and u_5 after submitting query q_5. Now, all other queries that result to a click on any of these URLs are identified, i.e., queries in the set $\{q_1, q_2, q_3, q_6\}$. The URL vector for each of these four queries has at least one URL in common with the URL vector of q_5.

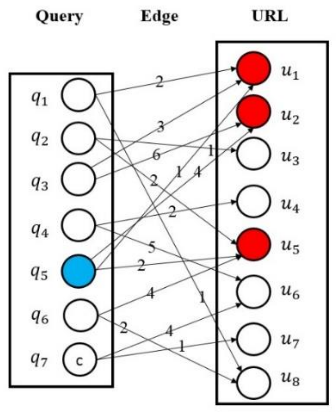

Fig. 2: A sample “Query-URL” bipartite graph

The similarity between q_5 and each query in the set $\{q_1, q_2, q_3, q_6\}$ is computed using Equation (2), while the rest of queries have zero similarity with q_5.
Since the average number of URLs clicked after each query is limited (e.g., about 3 in our dataset), computing the above similarity is not a difficult task. Although a click can be regarded as an indirect indication of a user’s interest to the content of the corresponding URL, yet clicking on a URL can not necessarily denote the relevancy between queries. For example, a user may click on a URL by mistake or face with a click bait, while having no interest to that content. On the other hand, there are many relevant queries that have no common URL clicks and thus their similarity becomes zero, especially when we know that usually users click on a few top results. An additional feature is required to detect similar queries despite differences in their texts and clicks.

3-2-3- Top-k Search Results and their Ranks Feature

The third feature, for computing similarity between queries, is related to the top-k search results of query returned by the search engine. The idea is if two queries have several common search results, it is highly probable that they are relevant, even if they have no common terms or clicks. We only consider the results of the first search result page which typically contains 10 results (i.e., k=10). This is due to the fact that the first page usually receives much more traffic (click rate) than other pages: According to a recent research on Google search engine, the first page attracts 91.5%, the second page 4.8%, the third page 1.1%, and the forth page 0.4% of the traffic [29].

The ranking of search results is also an important factor. For example, by submitting the query "apple" in Google, a couple of first search results are related to the Apple Inc. and the next results are related to the apple fruit, while for the query "apples", the top results are related to the apple fruit. Although these two queries are lexically similar (after stemming), the intent of users for submitting them is quite different. This sample shows why the search results of queries and their ranks are beneficial to identify similar queries. In cases where users do not click on the same results or even do not click on any search result, this feature can be a proper measure to compute similarity between queries. We first weigh the search results of a query according to their ranks. The higher the rank of a URL is, the more its weight would be. The reason for this weighting is that the search engines rank the URLs based on their relevance to the submitted query; and a higher ranked URL receives more clicks from users [12, 30]. Therefore, the value of a URL decreases as its rank increases. We define the weight of a URL with rank i, with $w[i] = \frac{1}{2^i}$.

As before, we construct a result vector for each query in which a URL u exists in the vector of query q (with weight of $w[i]$), if u is the i^{th}-ranked search result for query q. To compute the similarity between two queries q_x and q_y, the following equation is used:

$$Sim_{top-10}(q_x, q_y) = \frac{1}{2}\sum_{i=1}^{m} \frac{w[r_x(i)] + w[r_y(i)]}{|r_x(i) - r_y(i)| + 1} \qquad (3)$$

where m is the number of common URLs in the top 10 search results of two queries q_x and q_y, $r_x(i)$ and $r_y(i)$ are the ranks of the i^{th} common URL with respect to query q_x and q_y, respectively. According to the above formula, as much as two queries have more common search results with similar ranks (especially in top ranks), they get a higher similarity score. For example, the presence of a same search result l in the first rank of both q_x and q_y implies a much higher score, compared with the case where the result l appears in different ranks for two queries. According to the weight value we assign to a search result in rank i, high-ranked common results can significantly boost the similarity score. As we explained before, the rationale is that the rank of each URL indicates the amount of its relevance to the submitted query.

3-2-4- Final Similarity

To get the final similarity between two given queries q_x and q_y , the values of the above three features are combined linearly as follows:

$$Similarity\,(q_x, q_y) = \alpha * Sim_{ngrams}(q_x, q_y) + \beta * Sim_{bigraph}(q_x, q_y) + \gamma * Sim_{top-10}(q_x, q_y) \qquad (4)$$

The sum of α, β and γ parameters is equal to one (i.e., $\alpha + \beta + \gamma = 1$). Therefore, the result of Equation (4) is always between 0 and 1, as the value of all three features is also between 0 and 1. The setting of these hyper-parameters is explained in Section 4. In order to reduce the number of incorrect recommendations, we use a minimum similarity threshold θ such that if the final similarity between two queries is smaller than θ, we assume they have a zero similarity, i.e., they are irrelevant. Based on our experiments, we use $\theta = 0.01$.

3-2-5- Medoid-based Clustering

In the last step of the offline phase, we use the final similarity feature to cluster our training queries with a *k*-medoids algorithm. In contrast to the *k*-means algorithm which calculates the means of points as centroids, *k*-medoids chooses points themselves as centroids. This is more compatible with our notion of pairwise similarity defined in Equation (4), since we do not assign any feature to each individual point (i.e., two queries cannot be averaged, they can only be compared with respect to the similarity features).

The most common realization of *k*-medoids is the Partitioning Around Medoids (PAM) algorithm [31] which works effectively for small datasets, but does not scale well for large datasets due to its time complexity. There have been some efforts in developing fast algorithms for *k*-medoids clustering, e.g., with sampling and randomization techniques. In this paper, we use a simple and fast algorithm for *k*-medoids clustering that works in three steps [32]:

1. Select initial medoids: Calculate the distance between every pair of training points, select *k* most middle points as initial medoids using a local heuristic method in [32].

2. Update medoids: Find a new medoid of each cluster, which is the point minimizing the total distance to other points in its cluster.

3. Assign objects to medoids: Assign each point to the nearest medoid, calculate the sum of distances from all points to their corresponding medoids. If the sum is equal to the previous one, then stop the algorithm. Otherwise, go back to Step 2.

The above algorithm can take a significantly reduced time in computation with comparable performance, against the PAM [32]. We use this algorithm to cluster all training queries, using the inverse of final similarity in Equation (4) as the distance between two queries:

$$Distance\ (q_x, q_y) = \frac{1}{Similarity\ (q_x, q_y)} \qquad (5)$$

After clustering, we omit clusters containing only one query. As a result, at most 5% of queries are removed from our dataset, i.e., we have no recommendation for them. The remaining clusters may comprise a lot of queries which are not necessarily pairwise relevant, but it is very likely that they belong to the same category of information. For example, all queries about films, actors, and actresses may rest in a single cluster. Based on detected clusters, we construct three reverse lookup tables which will be used later to identify the corresponding cluster of an online (test) query:

- **N-gram lookup table:** For each 1/2/3-gram g and each cluster c, it shows how many times g is repeated in queries of cluster c.
- **Click lookup table:** For each URL u and each cluster c, it represents how many times u is clicked following queries in cluster c.
- **Result lookup table:** For each URL u and each cluster c, it indicates how many times u is returned as a search result of queries in cluster c.

The update of clusters is done periodically, e.g., in a per-day or per-week basis, through re-executing the above algorithm on the updated set of queries. During the update process, we can boost training queries based on the time of their occurrences, to put more emphasis on the recommendation of recent queries. We can also remove queries that have not been repeated for a while from the dataset.
It is worth mentioning that since we run the clustering algorithm in an offline environment, its running time does not affect the response time of test queries. In other words, we try to perform most time-consuming and complicated tasks in an offline manner to make our recommendations for online queries as soon as possible.

3-3- Online Randomized Query Recommendation

Our model is now trained and ready for query recommendation. In the online phase, when a test query q is submitted to the system, the closest cluster to q is found using the inverse lookup tables built in the clustering step. To do so, we first create an N-gram vector containing all 1/2/3-grams in q and also a URL vector having the first 10 search results of q. For each element in the N-gram vector, we search the N-gram lookup table and determine the frequency of occurring g in queries of cluster c as the score of cluster c. For the purpose of normalization, all cluster scores are divided by the maximum score. Similarly, for elements in the URL vector, we separately search the click and the result lookup tables to get the corresponding scores. The closest cluster to query q is the one with the highest total score with respect to the sum of above three scores. In rare cases where the score of top clusters are very close, we can select 2 or 3 clusters as the closest clusters to find relevant queries within them.
After finding the closest cluster which we call it designated cluster or dc in brief, we extract n most similar queries, in terms of the final similarity feature, from that cluster. In our work, we find top 10 similar queries (i.e., $n = 10$), and if the similarity between q and any of these queries is below the threshold θ, we ignore that query. To incur an acceptable query response time, we have to extract similar queries in a timely fashion. Since the number of queries in cluster dc can be large, calculating the similarity between q and every query in dc may be costly. We here devise a randomized nearest neighbor algorithm for approximating n most similar queries in regard to query q. This algorithm works iteratively as follows:

1. Initialization: Randomly select n queries from the designated cluster dc and put them in set S as the set of seed queries.
2. Distance calculation: Calculate the distance between q and every query in S, based on the distance metric in Equation (5).
3. Candidate selection: Sort all queries in S in ascending order, based on their distance from q, and put the first n queries in new set C (as the set of candidate queries).
4. Exploration: If new set C is different from the old C (obtained in the previous iteration), then for each query p in new C, greedily add n most similar queries with respect to p from cluster dc to set S and go to Step 2 for the next iteration. Otherwise, return new C.

The exploration step in the above algorithm is straightforward, as we do not perform any distance calculation or even sorting in this step. This is because the distance between every pair of queries in each cluster is calculated in the offline phase and we exactly know a priori which queries are the most n similar queries with respect to a query p in the designated cluster.
The advantage of the above algorithm is that it performs Step 2 and 3 for a limited number of queries in the designated cluster. In the worst case, it may go over all queries in dc or may return n queries that are not necessarily most similar to q. However, because of both randomized and greedy natures of the algorithm, it usually finds most similar queries after 2 or at most 3 iterations, even for very large clusters that contain thousands of queries. The results of our experiments on this algorithm is presented in the next section.

4- Evaluation

In this section, we explain the details of our experiments and present the evaluation results. All recommendation methods are implemented in Java with its native data structures such as ArrayList and LinkedList. To speed up the similarity computation and clustering process, we load all data directly in RAM without using any traditional databases. Our experiments are conducted on a server equipped with an Intel® Xeon® Processor X5650 and 32GB DDR3 RAM.

4-1- Experimental Dataset

To assess the efficiency of the recommendation algorithms, we use two separate datasets, from Parsijoo (the first Persian search engine) and AOL search engine logs1. The Parsijoo dataset consists of 7-days log of user activities from May 12 till May 18, 2018 and AOL search engine log contains web queries in a period of one month between March 1 and March 31, 2006. The statistics of two datasets are presented in Table 1. There is no record belonging to DDoS attacks in the Parsijoo log, as these attacks are abandoned by the front firewall. Moreover, we remove spam queries and click spams from our log as much as possible [33, 34].

Table 1: Statistics of Parsijoo and AOL datasets

Dataset	Time period	Number of queries	Number of queries led to a click	Number of unique queries
Parsijoo	7 days	932394	521542	246450
AOL	31 days	12138555	6484875	3382951

4-2- Evaluation Model

Our evaluation model is similar to the model used in previous works [17]. We first randomly partition our datasets into two roughly equal-sized parts and then, analyze the tuning of hyper-parameters, including α, β and γ in Equation (4), using the first part as the validation dataset. Then, we evaluate different algorithms using the other part as the evaluation dataset. We randomly select 1000 queries from the evaluation dataset as test queries and all remaining queries are used as training queries to train the model and cluster queries2. We perform the above steps for Parsijoo and AOL datasets separately. In order to see how robust our method is, we use different ratios of evaluation dataset (i.e., 90%, 70%, 50% and 30%) for training. For instance, when R=70%, we only inject 70% of the evaluation dataset for k-medoids clustering (the default ratio is R=90%). Also, the default number of recommendations is n=10.

As the evaluation metric, we use the precision which is defined as follows:

$$precision = \frac{TP}{TP+FP} \quad (6)$$

where TP (True Positives) denotes the number of relevant queries that are correctly recommended as relevant, and FP (False Positives) indicates the number of irrelevant queries that are wrongly recommended as relevant. In order to count TP and FP, we manually investigate the top results of recommendation algorithms for each test query to find out how many of them are relevant and irrelevant, respectively.

We also use the important p@10 metric as the number of correct recommendations made for each query over $n = 10$:

$$p@10 = \frac{TP}{10} \quad (7)$$

To understand the difference between two metrics, consider an example where a recommendation algorithm returns just one query and this query is relevant. The precision is 1, while the p@10 is 0.1. The results of precision and p@10 are averaged over all test queries. The running time and memory usage of the proposed method is calculated according to the executions on test queries.

4-3- Evaluation Results

4-3-1- Tuning of Parameters

To assess the impact of three hyper-parameters in Equation (4) and get the best configuration, we use an exhaustive grid search in the range [0,1] for α, β and γ parameters with step size of 0.1. We use the validation dataset for hyper-parameter tuning. Table 2 shows the results of precision and p@10 obtained in our method with different values of α, β and γ for the Parsijoo dataset. We separately tune our method for the AOL dataset as well. The highest precision and p@10 is achieved at $\alpha = 0.5$, $\beta = 0.3$ and $\gamma = 0.2$. For the AOL dataset, the best configuration is gotten at $\alpha = 0.4$, $\beta = 0.3$ and $\gamma = 0.3$. The N-gram feature reflects the pure textual similarity between queries and hence, it has a higher coefficient than other two features. On the other hand, the bipartite graph feature is more important than the top-10 search results feature, since it is supported by users' clicks. Thus, we can generally conclude that $\alpha > \beta \geq \gamma$. It is interesting that whenever we ignore any of these three features, we get a very bad result, especially with regard to the p@10 metric. For example, the p@10 degrades by up to 45% in cases where one of α, β and γ parameters is set to 0, while the precision worsens about 15%.

[1] The 3-months search log of AOL is publicly available over the Internet, e.g., in [8].
[2] To remove noises, we only consider test queries that have been repeated at least 5 times by different users.

Table 2: The precision and p@10 results of our method obtained with different values of α, β and γ parameters from Parsijoo dataset

α	β	γ	precision	p@10
0.5	0.3	0.2	66.8%	63.9%
0.4	0.3	0.3	65.4%	62.7%
0.5	0.5	0	61.3%	54.5%
0.5	0	0.5	62.1%	57.5%
0	0.5	0.5	57.9%	48%
1	0	0	60.2%	47.6%
0	1	0	54.3%	42.3%
0	0	1	55.4%	33.1%

Although the tuning of hyper-parameters is dependent on the dataset, but their optimum setting is not influenced by choosing different ratios of evaluation dataset (R). More specifically, the best value of α, β and γ is roughly the same for two cases where R=90% and R=30%.

Fig. 3 shows the impact of choosing n, i.e., the number of recommendations, on the precision (n varies from 1 to 10). We conduct this experiment on both Parsijoo and AOL datasets. The best result is achieved at $n = 1$. As the number of recommendations increases, the recommended queries become less similar to the user's submitted query, from the view point of three features, and thus, the number of false positives grows. The precision decreases about 30% when we change n from 1 to 10. For large values of n, the precision does not change much, as our algorithm can rarely add any more recommendation for a significant portion of queries (e.g., the precision at $n = 9$ is only 1% higher, compared with $n = 10$). The precision of recommendation for AOL is on average 8% higher than that of Parsijoo, especially for larger values of n. The main reason behind this phenomenon is that the amount of queries and users in the AOL dataset is an order of magnitude larger than queries in Parsijoo. Thus, the likelihood of finding relevant queries for a test query is higher.

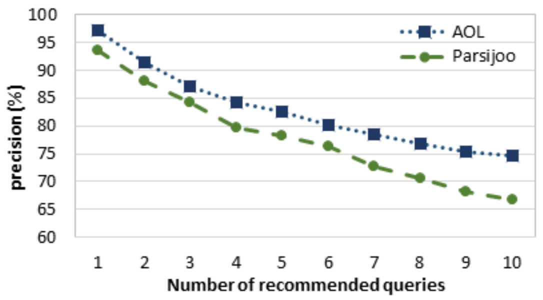

Fig. 3: The precision results of our method with respect to number of reommendations (n), obtained from AOL and Parsijoo datasets

Now, we analyze how the volume of training dataset affects the test results. The result of precision and p@10 for different ratios of Parsijoo dataset are shown in Fig. 4. As the amount of training data shrinks, the precision declines slightly. For example, the precision worsens only 11%, when R decreases from 90% to 30%. As much as we reduce the size of training dataset, the number of true relevant queries recommended for a given test query decreases. However, since we use a minium similarity threshold between relevant queries in Equation (4), the number of irrelevant queries recommended by our algorithm does not change much. But, the story is different when we pay attention to the p@10 metric. It is affected considerably by the value of R in a way that it drops by about one third when R falls to 30%. This is because our algorithm can barely find multiple relevant queries as we have not enough training queries. The trend of results in Fig. 4 remains roughly the same for AOL dataset. However, since the number of training queries is much larger in this dataset, the decrease in p@10 is not so devastating (the recall gets halved, as we reduce R from 90% to 30%).

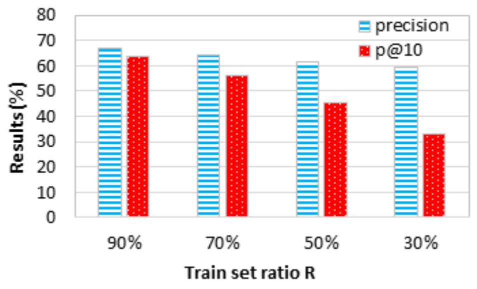

Fig. 4: The evaluation results of our method with respect to different evaluation dataset ratio *R*

The number of clusters in the *k*-medoids clustering algorithm impacts on both precision and response time of our method. Fig. 5 demonstrates the evaluation results of the proposed method obtained for Parsijo dataset when *k* raises from 50 to 500 with step size of 50. For smaller values of *k*, the precision is lower because it is more likely that irrelevant queries are placed in the same cluser. As a result, our randomized nearest neighbor algorithm probably returns some irrelevent queries in response to a submitted query. As much as we enlarge *k*, the precision improves as well. However, this improvement becomes negligible for very large values of *k*. On the other hand, the p@10 metric worsens when *k* goes beyond 350, because the number of relevant queries in clusters is reduced and our randomized algorithm fails in many cases to find 10 similar queries. Based on these results, we choose *k*=400 as the optimum value for our experiments.

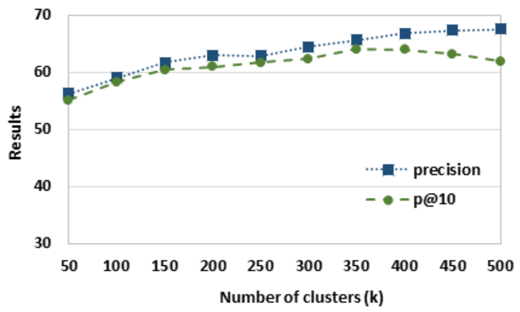

Fig. 5: The evaluation results of our method with respect to different number of clusters (*k*), obtained from AOL and Parsijoo datasets

The choice of *k* depends on the properties of dataset. Thus, we conduct the same experiment on the AOL dataset which leads to *k*=900 as the best choice. It is fascinating that the tunning of *k* is independent of the dataset ratio R (if the training queries are randomly selected from the whole dataset). From the perspective of response time, our randomized algorithm works faster when we increase the number of clusters, because it explores all candidate queries in fewer iterations. For example, the response time of the online algorithm gets halved when we change *k* from 50 to 500.

4-3-2- Evaluating the Online Algorithm

As discussed before, the randomized algorithm, proposed for approximating most similar queries in the online phase, makes a good tradeoff between the performance and response time of the recommendation system. We compare the performance results of this algorithm with the exhaustive algorithm which exactly finds the most similar queries through brute-force search in the designated cluster. The results of this comparison obtained from the Parsijoo dataset are presented in Table 3. The randomized algorithm produces the response 3 times faster than the exhaustive one at the cost of only 2% reduction in precision. As shown in the table, the average number of candidate queries investigated to find most similar queries is reduced by a factor of about 4. The results are improved further for bigger datasets like AOL, namely the response time decreases by a factor of 6 with only 3% loss in precision.

Table 3: Comparison between exhaustive and randomized recommendation algorithms

Algorithms	precision	p@10	#candidate queries	response time
Exhaustive nearest neighbor	68.1%	65.3%	672	615ms
Randomized nearest neighbor	66.8%	63.9%	164	201ms

4-3-3- Comparing Different Algorithms

Finally, we compare the performance of our method with three famous query recommendation algorithms introduced in [17], [13] and [24], which we refer to by “Bipartite graph”, “Query clustering” and “KB-QREC”, respectively. Table 4 shows the results of precision and p@10 obtained separately from Parsijoo and AOL datasets. Our proposed method, has a better precision in comparison with the other three methods, especially when we conside the Parsijoo dataset. More precisely, the proposed method outperforms KB-QREC by about 7%, and overcomes the bipartite graph and query clustering methods by more than 9% and 20%, respectively. From the viewpoint of p@10, the gap between the results of our method and other methods becomes even more apparent in a way that the improvement reaches to at least 23%. We achieve this advantage through utilizing various kinds of features simultaneously (from a linguistic feature to a user behavioral feature) to find similar queries. This property is more vital when we consider smaller datasets, since the only way to generate multiple recommendations is to catch similar queries from all different perspectives.

Table 5 compares the above four query recommendation methods with respect to the response time and memory usage parameters, obtained from Parsijoo and AOL datasets. To get the results, we submit 1000 queries to each recommendation method and then measure these two parameters for each method separately. First of all, since the size of AOL data is much larger than that of Parsijoo, we observe a considerably higher value for both parameters in all methods, when we take this dataset into account. The Bipartite graph is the simplest algorithm which is solely based the query-click graph. As a result, it yields the minimum value for both response time and memory usage. On the other hand, our method works with three different kinds of information to compute similarity features and hence, it has the worst memory usage. For instance, with Parsijoo dataset, it consume about 10%, 20% and 50% more memory than KB-QREC, Query clustering and Bipartite graph, respectively. This gab expands even further, when we consider the bigger AOL dataset.

Table 4: Comparison of different methods with respect to precision and p@10

Methods	Parsijoo		AOL	
	precision	p@10	precision	p@10
Query clustering [13]	51.5%	43.1%	62.2%	59.1%
Bipartite graph [17]	61.4%	38.8%	69.6%	62.6%
KB-QREC [24]	62.7%	48.8%	70.8%	67.2%
Our proposed method	66.8%	63.9%	74.7%	72.2%

In regard to the response time factor, the story becomes completely different. With Parsijoo dataset, the response time of our algorithm is about 10% and 16% shorter than KB-QREC and Query clustering methods, respectively (it still incurs a 10% longer response time than Bipartite graph). The response time of our method (in comparison with other methods) is improved more, when using AOL dateset. More precisely, it outperforms KB-QREC and Query clustering by about 25% and 40%, respectively. This advantage is mainly due to the fast randomized algorithm we use in the online phase of our proposed method, as it tries to keep the set of candidate queries for finding most similar ones as small as possible.

Table 5: Comparison of different methods with respect to system parameters

Methods	Parsijoo		AOL	
	response time	memory usage	response time	memory usage
Query clustering [13]	386ms	51MB	1437ms	575MB
Bipartite graph [17]	288ms	44MB	789ms	409MB
KB-QREC [24]	354ms	56MB	1096ms	603MB
Our proposed method	321ms	62MB	814ms	681MB

5- Conclusions

In this paper, a novel method is proposed for query recommendation in search engines, using a combination of two query clustering and graph modeling approaches. As opposed to former methods, we take into account diverse features (query, click and result) to define similarity between queries and cluster them. In order to improve both precision and response time, we use an efficient *k*-medoids clustering algorithm as well as a new randomized nearest neighbor algorithm to find most similar queries. Evaluation results show that the proposed method outperforms some famous query recommendation methods with respect to precision and p@10 metrics. For example, the p@10 is improved by at least 23%, when compared with other counterparts.

References

[1] J. Wang, J. Z. Huang, J. Guo, and Y. Lan, “Query Ranking Model for Search Engine Query Recommendation”, International Journal of Machine Learning and Cybernetics, Vol. 8, No. 3, 2015, pp. 1-20.

[2] M. Mitsui, “A Generative Framework to Query Recommendation and Evaluation”, in ACM Conference on Human Information Interaction and Retrieval (CHIIR), 2017, pp. 407-409.

[3] X. Zhu, J. Guo, X. Cheng and Y. Lan, “More than Relevance: High Utility Query Recommendation by Mining Users' Search Behaviors”, in 21st ACM International Conference on Information and Knowledge Management (CIKM), 2012, pp. 1814-1814.

[4] P. Melville and V. Sindhwani, “Recommender Systems”, Encyclopedia of Machine Learning, Springer US, pp. 829-838, 2011.

[5] Y. Liu, J. Miao, M. Zhang, S. Ma and L. Ru, “How Do Users Describe Their Information Need: Query Recommendation based on Snippet Click Model”, Expert Systems with Applications, Vol. 38, No. 11, 2011, pp. 13:847-13:856.

[6] W. Song, J. Z. Liang, X. L. Cao and S. C. Park, “An Effective Query Recommendation Approach using Semantic Strategies for Intelligent Information Retrieval”, *Expert Systems with Applications*, Vol. 41, No. 2, 2014, pp. 366-372.

[7] Y. Song, D. Zhou and L. W. He, “Query Suggestion by Constructing Term-transition Graphs”, in 5th ACM International Conference on Web Search and Data Mining (WSDM), 2012, pp. 353-362.

[8] Web Search Query Log Downloads. https://www.jeffhuang.com/search_query_logs.html (2018).

[9] Z. Zhang and O. Nasraoui, “Mining Search Engine Query Logs for Query Recommendation”, in 15th International Conference on World Wide Web (WWW), 2006, pp. 1039–1040.

[10] J. R. Wen, J. Y. Nie and H. J. Zhang, “Clustering User Queries of a Search Engine”, in 10th International Conference on World Wide Web (WWW), 2001, pp. 162-168.

[11] R. Baeza-Yates, C. Hurtado and M. Mendoza, “Query Recommendation using Query Logs in Search Engines”, in International Conference on Extending Database Technology (EDBT), 2004, pp. 588-596.

[12] Y. Hong, J. Vaidya and H. Lu, “Search Engine Query Clustering using Top-k Search Results”, in IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011, pp. 112-119.

[13] R. Chaudhary and N. Taneja, “A Novel Approach for Query Recommendation via Query Logs”, International Journal of Scientific and Engineering Research (IJSER), Vol. 3, No. 8, 2012, pp. 1-6.

[14] L. Fitzpatrick and M. Dent, “Automatic Feedback using Past Queries: Social Searching?”, in 20th Annual International Conference on Research and Development in Information Retrieval (SIGIR), 1997, pp. 306–313.

[15] K. Sugiyama, K. Hatano and M. Yoshikawa, “Adaptive Web Search based on User Profile Constructed without any Effort from Users”, in 13th Conference on World Wide Web (WWW), 2004, pp. 675–684.

[16] B. J. Chelliah, R. Ojha, S. Semwal, P. Dobhal and C. Sahu, “Personalized Search Engine with Query Recommendation and Re-ranking”, Journal of Network Communications and Emerging Technologies (JNCET), Vol. 8, No. 4, 2018, pp. 213-217.

[17] Q. Mei, D. Zhou and K. Church, “Query Suggestion using Hitting Time”, in 17th ACM Conference on Information and Knowledge Management (CIKM), 2008, pp. 469–478.

[18] P. Boldi, F. Bonchi, C. Castillo, D. Donato, A. Gionis and S. Vigna, “The Query-flow Graph: Model and Applications”, in 17th ACM Conference on Information and Knowledge Management (CIKM), 2008, pp. 609-618.

[19] A. Anagnostopoulos, L. Becchetti, C. Castillo and A. Gionis, “An Optimization Framework for Query Recommendation”, in 3rd ACM International Conference on Web Search and Data Mining (WSDM), 2010, pp. 161-170.

[20] L. Bai, J. Guo and X. Cheng, “Query Recommendation by Modelling the Query-flow Graph”, in Asia Information Retrieval Symposium (AIRS), 2011, pp. 137-146.

[21] F. Bonchi, R. Perego, F. Silvestri, H. Vahabi and R. Venturini, “Recommendations for the Long Tail by Term-Query Graph”, in 20th International Conference on World Wide Web (WWW), 2011, pp. 15-16.

[22] J. Wang, J. Z. Huang, D. Wu, J. Guo and Y. Lan, "An Incremental Model on Search Engine Query Recommendation", Elsevier Neurocomputing, Vol. 218, 2016, pp. 423-431.
[23] A. Sordoni, Y. Bengio, H. Vahabi, C. Lioma, J. G. Simonsen and J. Y. Nie, "A Hierarchical Recurrent Encoder-decoder for Generative Context-aware Query Suggestion", in 24th ACM International Conference on Information and Knowledge Management (CIKM), 2015, pp. 553-562.
[24] Z. Huang, B. Cautis, R. Cheng and Y. Zheng, "Kb-enabled Query Recommendation for Long-tail queries", in 25th ACM International Conference on Information and Knowledge Management(CIKM), 2016, pp. 2107-2112.
[25] R. Cheng, Y. Zheng and J. Yan, "Entity-based Query Recommendation for Long-tail Queries", ACM Transactions on Knowledge Discovery from Data, Vol. 1, No. 1, 2018, pp. 1-22.
[26] S. Kannan and V. Gurusamy, "Preprocessing techniques for text mining", International Journal of Computer Science & Communication Networks, Vol. 5, No. 1, 2014, pp. 1-7.
[27] Hunspell GitHub. https://github.com/hunspell (2018).
[28] H. Taghi-Zadeh, M. H. Sadreddini, M. H. Diyanati, and A. H. Rasekh, "A New Hybrid Stemming Method for Persian Language", Digit. Scholarsh. Humanit., Vol. 32, No. 1, 2017, pp. 209–221.
[29] Infront Webworks: Value of Organic First-page Results. https://www.infront.com/blogs/the-infront-blog/2015/6/17/value-of-first-page-google-results (2018).
[30] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski and G. Gay, "Evaluating the Accuracy of Implicit Feedback from Clicks and Query Reformulations in Web Search", ACM Transactions on Information Systems (TOIS), Vol. 25, No. 2, 2007, pp. 183-190.
[31] L. Kaufman, and P.J. Rousseeuw, "Clustering by Means of Medoids in Statistical Data Analysis based on the L1–norm and Related Methods", North-Holland, pp. 405–416, 1987.
[32] H. S. Park, and C. H. Jun, "A Simple and Fast Algorithm for K-medoids Clustering", Elsevier Expert Systems with Applications, Vol. 36, No. 2, 2009, pp. 3336-3341.
[33] T. Shakiba, S. Zarifzadeh, and V. Derhami, "Spam Query Detection using Stream Clustering", Springer World Wide Web (WWW), Vol. 21, No. 2, 2018, pp. 557–572.
[34] M. Fallah, S. Zarifzadeh, "Practical Detection of Click Spams using Efficient Classification-based Algorithms", International Journal of Information and Communication Technology Research, Vol. 10, No. 2, 2018, pp. 63-71.
[35] C. D. Manning, P. Raghavan and H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.

Elham Esmaeeli-Gohari received the B.S. degree in Computer Engineering from Shahid Bahonar University, Kerman, Iran in 2013, and M.S. degree in Software Engineering from Yazd University, Iran, in 2017. Currently she is a Ph.D. Student in Isfahan University, Iran. Her research interests include Autonomous vehicles, Data fusion, Big data, Recommendation systems, Machine learning and Data mining.

Sajjad Zarifzadeh is an assistant professor of Computer Engineering at Yazd University, Iran. He received his Ph.D. in Computer Engineering from University of Tehran, Iran in 2012. His research is focused on Data analysis, Big data, Network security, Web and Internet services.