RAF: An Activation Framework for Refining Similarity Queries Using Learning Techniques

Yiming Ma, Sharad Mehrotra, Dawit Yimam Seid, and Qi Zhong

Department of Computer Science, University of California, Irvine, USA
{maym, sharad, dseid, qzhong}@ics.uci.edu

Abstract. In numerous applications that deal with similarity search, a user may not have an exact specification of his information need and/or may not be able to formulate a query that exactly captures his notion of similarity. A promising approach to mitigate this problem is to enable the user to submit a rough approximation of the desired query and use relevance feedback on retrieved objects to refine the query. In this paper, we explore such a refinement strategy for a general class of structured similarity queries. Our approach casts the refinement problem as that of learning concepts using the tuples on which the user provides feedback as a labeled training set. Under this setup, similarity query refinement consists of two learning tasks: learning the structure of the query and learning the relative importance of query components. The paper develops machine learning approaches suitable for the two learning tasks. The primary contribution of the paper is the Refinement Activation Framework (RAF) that decides when each learner is invoked. Experimental analysis over many real life datasets shows that our strategy significantly outperforms existing approaches in terms of retrieval quality.

1 Introduction

With the proliferation of the web and emergence of applications requiring flexible search over diverse data types, effective support for personalized similarity search in database systems has emerged as an important research challenge. Similarity based retrieval systems are also increasingly used for exploratory data analysis and retrieval where a user may not initially have a clear mental model of his exact information need [1]. A promising approach to overcome the "initial query" and "subjectivity" problems is that of automatic query refinement via user feedback. In such an approach, a user starts with an approximate initial query and communicates his preferences to the system by providing feedback (judgments on the relevance or quality of answers). The system then modifies the query internally (e.g., changes the levels of importance of the different search criteria, and adds/removes search criteria) to better focus on the distinguishing features of tuples deemed relevant. The modified query is re-evaluated and the cycle of refinement continues until the user is satisfied with the results.

Query refinement via feedback has been explored extensively in the context of text document retrieval [16, 1]. More recently, its effectiveness in feature-based image and multimedia similarity retrieval [14, 18, 7] as well as similarity search over metric spaces [20] has also been established. Authors in [13] considered the problem of refining SQL queries from user interactions in an object relational database. This paper explored a limited set of SQL queries and illustrated that even simple extensions of refinement approaches studied in the IR literature can significantly enhance users' search experience.

This paper considers the problem of refining a general class of SQL queries. In contrast to the ad-hoc approaches in [13], we postulate the SQL refinement problem as that of concept learning from examples to which many existing machine learning solutions can be applied. A direct (naive) strategy to modify the query is to view the records on which the user provides feedback as a labeled training set and use a classifier (e.g., decision tree [15]) to learn a new query representation that will replace the original query. However, such a naive strategy does not perform well in practice. Reasons for this include the fact that classifiers do not work well when the training set is very small as is the case in our setting where a user may provide feedback on only a few records. Furthermore, being purely based on training data, the naive strategy ignores the initial query provided by the user. Consequently it may get trapped in a wrong hypothesis simply because the hypothesis fits the few examples. In addition, it is very difficult to incorporate user defined types and similarity functions into existing classifiers. Consistency is also an important issue in the refinement task. Many existing classifiers are sensitive to the inputs causing the models built at different refinement iterations to excessively differ from each other.

We view query refinement from feedback as consisting of two interrelated learning tasks – learning the query structure, and learning the query weights that capture the relative importance of different query components. These two learning tasks have different motivations and serve different purposes. For instance, structural changes to the query are very useful when the initial user query is incomplete which may happen if either a user is not familiar with the database or finds it too laborious to postulate a proper query. In contrast, weight adjustment serves the purpose of customizing/tuning the ranking of results to reflect the subjective importance of the different query components to the user. Thus, determining when to invoke the structure/weight learners becomes a key issue in refining a query. We develop a formal basis for making such a decision based on which the *Refinement Activation Framework (RAF)* is designed.

The main contributions of this paper are summarized as follows:

1. We provide a powerful framework for refining a general class of SQL similarity queries that uses a multi-modal refinement activation procedure to adjust both query predicate weights and query structure using learning techniques.
2. We provide and experimentally validate a novel query structure refinement technique that is based on a decision tree learner.

The remainder of the paper is organized as follows. Section 2 describes background concepts that form the basis of our work. Section 3 discusses our approach to query refinement. Section 4 presents our experiments. Section 5 discusses related work. We conclude with Section 6.

2 Background

2.1 Similarity Queries

A similarity query consists of three components: a set of similarity predicates structured in DNF form, a set of weights assigned to each similarity predicate and a ranking function. In this section, we describe these components.

The search condition in a similarity query is represented as a Boolean DNF (Disjunctive Normal Form) expression over similarity predicates. Formally a query $Q = C_1 \vee C_2 \vee \ldots \vee C_n$ is a DNF expression where $C_i = C_{i1} \wedge C_{i2} \ldots, C_{in}$ is a conjunction, and each C_{ij} is a similarity predicate. A **similarity predicate** is defined over a domain of a given data type. It is a function with two inputs: (1) an attribute value from a tuple, t, (2) a target value which can be a point or a range. It returns a similarity *score* in the range [0,1]. Notice that restricting ourselves to DNF queries does not limit the generality of our approach since any query condition can be mapped to its DNF representation. As will become evident later, using a DNF representation facilitates structural learning.

A **DNF ranking function** is a domain-specific function used to compute the score of a tuple by aggregating scores from individual similarity predicates according to the DNF structure of a search condition and its corresponding set (template) of **weights** that indicate the importance of each similarity predicate. The *template of weights* matches the structure of the search condition and associates a weight to each predicate in a conjunction and also to each conjunction in the overall disjunction.

A DNF Ranking Function first uses *predicate weights* to assign aggregate scores for each conjunction, and it then uses *conjunction weights* to assign an overall score for the query(disjunction). We aggregate the similarity scores of predicates in a conjunction with a weighted L_1 metric (weighted summation). Using weighted L_1 metric as a conjunction aggregation function has been widely used in text IR query models where a query is typically expressed as a single conjunction [16, 1]. To compute an overall score of a query (disjunction), we use the MAX function over the weighted conjunction scores. MAX is one of the most popular disjunction aggregation functions [4]. The weight learning algorithm used in this paper is optimized for these settings. However, our overall refinement algorithm is extensible enough to take other alternative ranking functions [19, 12] as long as a weight learning module is properly defined.

All predicate weights in a conjunction add up to 1. All conjunction weights in a disjunction may not add up to 1. A conjunction weight is in the range of [0, 1], and represents the importance of the conjunction to the user. For example, a conjunction's importance can be measured as the percentage of relevant tuples covered by it. The final aggregated scores produced by a ranking function are used for ranking the tuples. A similarity query returns a set of records with $score \geq \alpha$ (also called α *cut* or *threshold*).

Example 1. Table 1 shows an example data table to be used throughout this paper to illustrate our approach. It is a contract job listing table containing four attributes: location, salary, employment duration, and job description. Consider

Table 1. Example data table

ID	Loc	Sal	Dur(yr)	Desc
1	SN	65K	1.5	DB Developer
2	LA	70K	1	DBA
3	SD	60K	1.5	DB Designer
4	SF	70K	1.5	DB Developer
5	...	...	...	...

Table 2. Example feedback table

ID	Loc	Sal	Dur	Desc	Feedback
1	SN	65K	1.5	DB Developer	OK
2	LA	70K	1	DBA	NOK
3	SD	60K	1.5	DB Designer	OK
4	SF	70K	1.5	DB Developer	OK
...	...	...	...	...	...

the following query that asks for jobs located near SN or that pay more than 65K and have a duration close to 2 years (expressed as a DNF search condition):

(LocNear (Loc, "SN") **AND** DurClose(Dur, 2)) **OR** (SalGreater (Sal, 65000) **AND** DurClose(Dur, 2))

This search condition can be directly implemented on top of a RDBMS that supports user-defined functions (UDFs) as follows:

```
WITH Score AS (SELECT ID,
        LocNear (Loc, "SN") AS l_s,
        DurClose(Dur, 2) AS D_s1,
        SalGreater (Sal, 65000) AS S_s FROM Job)
SELECT RankVal(W_1, Score.l_s, w_11, Score.D_s1, w_12,
      W_2, Score.S_s, w_21, Score.D_s1, w_22) AS S,
      Loc, Sal, Dur, Desc
      FROM Job, Score WHERE Job.ID=Score.ID AND S ≥ α
     ORDER BY S DESC
```

A corresponding weight template may be: $(W_1(w_{11}, w_{12}), W_2(w_{21}, w_{22})) = (0.9(0.4, 0.6), 0.8(0.4, 0.6))$. The ranking function *RankVal* uses this template to aggregate similarity scores as: $MAX[0.9 \times (0.4 \times l_s + 0.6 \times D_{s1}), 0.8 \times (0.4 \times S_s + 0.6 \times D_{s1})]$. The condition includes three similarity predicates: *LocNear*, *SalGreater* and *DurClose* with obvious semantics. For example, *SalGreater* may return 1 if $salary \geq 65K$. For $30K < salary < 65K$, it returns $\left(1 - \frac{|sal-65K|}{40K}\right)^r$, where r is an integer. It returns 0 if $salary < 30K$. Such functions are application specific and designed by domain experts and implemented by database developers as UDFs. As indicated by the higher weight assigned to the *DurClose* predicate, the user is more interested in a job that has duration close to 2 years than one that is close to SN or pays more than 65K. Although tuples with ID 1 and 4 receive the same conjunction scores from the first and second conjunction, the first tuple is ranked higher since the weight for the first conjunction is higher. Therefore, the tuple with ID 1 should be returned as the top result.

2.2 Similarity Query Refinement

Given a search condition q, a set r of top-k records returned by q, and relevance feedback f on these records (i.e., a triple $\langle q, r, f \rangle$), the similarity query refinement problem is to modify the search condition q in such a way that, when re-executed, it will rank more relevant records at the top. The interactive search process to find satisfactory answers to a particular query is called a *query session.* A query session can include one or more *refinement iterations* where the user provides feedback and the system refines the query based on the feedback and returns another set of ranked results. A query refinement in an iteration may involve adapting the predicates, the DNF condition structure as well as the weights.

Example 2. Before arriving at the condition given in example 1 that returns the best results, suppose the user started the query session with the following query that approximately captures his information need: $((LocNear(loc, \text{``}SN''))$ **AND** $DescClose(Desc, \text{``}DBA''))$, with a corresponding weight template of (1(0.5, 0.5)). At this point the user may, for example, be unfamiliar with the data stored in the database. As a result, he used "*Description* similar to 'DBA' " as one of the search conditions. Then, after seeing a few records returned by the initial query whose description matches "DBA" but whose other attributes do not match his interest, he may use relevance feedback to invoke a refinement iteration to guide the query away from this search condition eventually leading to its removal in the final query shown in example 1.

3 Learning Queries from User Feedback

3.1 Problem Formulation

For each query session we maintain two tables: an *answer table* which contains the ranked answer tuples along with the similarity scores assigned to these tuples, and a *feedback table* that stores the relevance feedback [1] given by the user on tuples in the answer table. Table 2 shows an example of such a table. The problem of query refinement can now be cast as a problem of utilizing *feedback table* to learn a classifier. In principle, any concept learning method can be employed provided that it performs well on a small number of examples. However, to be effective, query refinement requires a careful application of learning methods. In particular, simply replacing the original query with the newly learned query can have undesirable consequences. This leads to the important question of how to modify the original query based on the learned classifier.

In this section, we consider two different types of learning algorithms (classifiers). One focuses on query weight tuning, and another focuses on query structure tuning. An activation algorithm is used to control the overall learning process that consists of these two interrelated learning tasks.

3.2 Refinement Activation Framework (RAF)

Given a query and user feedback on its results, RAF determines which of the two learning task to invoke. Structural changes result in addition (deletion) of a new (old) conjunction to the DNF query or addition (deletion) of a new (old) predicate within a conjunction. In contrast to gradual weight adjustment that results in fine tuning of rank order, structural changes can dramatically change the return set even causing objects deemed irrelevant to the original query to be ranked at the top of the result set. Hence, structural changes to the query must be made conservatively since an incorrect change may lead a refinement along a wrong path. RAF invokes structural modifications only when

[1] For simplicity, we assume binary relevance judgments (i.e. "Relevant" or "Not Relevant") although our approach can also support finer grained distinctions.

ID	ConjID=1:LocNear (Loc,"SN")	ConjID=2:DescClose (Desc,"DBA")	Feed-back
1	1	0.8	OK
2	0.8	1	NOK
3	0.7	0.6	OK
4	0.5	0.8	OK

Fig. 1. Example CSS table

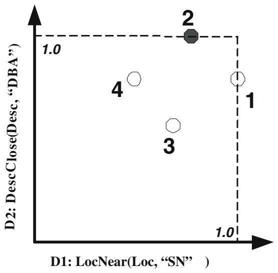


Fig. 2. CSS for Fig 1

feedback provided by the user requires that such modifications be made and the desired effect cannot be achieved by re-weighting query components. In order to identifying situations where structural modifications are needed, below we explore the limitations of weight learning in the context of refinement.

We begin by developing some notation. Consider a single conjunction $C_i = C_{i1} \wedge C_{i2} \ldots, C_{in}$ of a DNF query. Since each predicate (i.e., C_{ij}) in the conjunction returns a similarity value in the range of [0,1], together they form a *Conjunction Similarity Space* (CSS). Each dimension in the CSS represents a predicate and a tuple can be mapped to a point in the CSS. We store these mapped points in a table called *CSS table.* For example, figure 1 shows an example CSS table for the tuples in the feedback table shown in table 2. This space is also depicted in figure 2. Next, we define the notions of *CSS domination* and *CSS conflict* as follows:

Definition 1 (CSS domination). *In a given CSS, a tuple $t1$ dominates $t2$ if $t1$ is as good or better (that is, $t1$ has an equal or higher similarity value) in all dimensions and better in at least one dimension compared to $t2$.*

For example, in the Figure 2, tuple 2 dominates tuples 3 and 4, but does not dominate tuple 1. Similarly, tuple 1 dominates tuples 3 and 4, but not tuple 2.

Definition 2 (CSS conflict). *In a given CSS, a pair of feedback tuples ($t1$, $t2$) conflict with each other if $t1$ receives negative feedback, $t2$ receives positive feedback, and either $t1$ dominates $t2$ or $t1$ and $t2$ have equal values in all dimensions.*

The conflicts in the CSS space in Figure 2 are between tuples 2 and 3, as well as tuples 2 and 4, but not between tuples 2 and 1. Presence of conflicts in the return set of a query means that the query is ranking an irrelevant tuple higher than a tuple deemed relevant by the user. Hence, the query does not capture the user's information need and must be modified. Unfortunately, simply modifying weights associated with the predicates can not resolve the conflict as is stated in the following lemma.

Lemma 1. *For a given CSS, if there is a conflicting tuple pair ($t1$, $t2$), there is no monotonic aggregation function that can resolve this conflict by assigning a larger score to $t2$.*

Proof. From [5], an aggregation function f is *monotone* if $f(x_1, \ldots, x_m) \leq f(x'_1, \ldots, x'_m)$ whenever $x_i \leq x'_i$ for every i. Recall that we use weighted summation, which is monotonic, to aggregate scores in a CSS. Let $(t1, t2)$ be a conflicting tuple pair where $t1$ receives negative and $t2$ receives positive feedback. Since $t1$ has as good or better value in all dimensions, it is not possible to assign a bigger score to $t2$ since any weight assignment (done on predicates) applies to all tuples. Note that, even without the last condition in definition 2 (i.e., $t1$ and $t2$ have equal values in all dimensions), lemma 1 is still true. By adding this condition, we capture cases that could not be resolved by weight tuning □

We have, thus far, established the limitation of the weight learning approach in the context of refinement. Resolving conflicts requires structural modifications to the query. The activation procedure we develop utilizes the above observation to determine when to invoke such modifications.

Let $Q = C_1 \vee C_2 \vee C_n$ be a DNF query where $C_i = C_{i1} \wedge C_{i2} \ldots, C_{in}$ is a conjunction. To refine Q, first the activation procedure will decide how the feedback on each tuple is used for refinement. Specifically, it decides, for each tuple with a feedback, which conjunction should be refined by it. The activation procedure attaches feedback to different conjunctions by assigning each feedback to the highest scored conjunction[2]. For each conjunction C_i, it then determines if the assigned feedback contains any conflict. If a conflict is identified, structural modification to the query is invoked based on the conflicting set. For example, in Figure 2 where empty circles represent positive feedback, tuple 2 conflicts with tuples 3 and 4. To resolve these conflicts, the activation procedure invokes structural learning using tuples 2, 3, and 4. The structure learning algorithm will attempt to learn a predicate that, when added to the conjunction, will resolve the conflict. In the current example, it may suggest the addition of a predicate $Duration \geq 1.5$ to the conjunction. In general, depending on the mechanism used, the conjunction may be augmented by not only a single predicate but also a logical formula F consisting of multiple predicates connected by logical connectives. In the new formula (i.e., $newConj = C_i \wedge F$), most (or all) of the conflicts associated with the original conjunction can be resolved. We temporarily treat F as a pseudo-predicate so the new formula becomes a conjunction. The weights of predicates in this conjunction are rebalanced using a predicate weight learning method where, a predicate is dropped if its weight falls below a threshold. We designate the assigned weight of F by W_F.

In this paper, we assume a logic formula $F = F_1 \vee F_2 \vee \cdots \vee F_k$ is itself in DNF. The predicates in C_i of $newConj$ can be distributed over the F_p, resulting in a new candidate set of conjunctions $(C_i \wedge F_1) \vee (C_i \wedge F_2) \vee \cdots \vee (C_i \wedge F_k)$. Hence, when resolving conflict tuples in a conjunction C_i, we may potentially learn new conjunctions. During this process, the actual predicate weight of F_p in a conjunction is assigned to be W_F. We also use the original conjunction (C_i) weight to initialize the weights of candidate conjunctions (i.e., $C_i \wedge F_p$).

[2] Since the disjunction aggregation function is MAX, assigning a feedback tuple to the highest scored conjunction is appropriate. Different assignment schemes may be needed for other aggregation functions.

RAF-Algo()

```
1: Input: Query ($Q$), FeedbackTable ($FT$)
2: Output: New Query ($Q'$)
3: compute_CSS_tables($Q$, $FT$)
4: New Conjunctions: $newConjs = \emptyset$; $newConjsCunt = 0$
5: for each conjunction $C_i$ in a query $Q$ do
6:     $CSST$ = get_CSS_table($C_i$)
7:     $Conflicts$= computeConflictSet($CSST$)
8:     $F = \emptyset$
9:     if $|Conflicts| \geq 1$ then
10:        $F$= learnStructure($C_i$, $Conflicts$)
11:    $newConj$=$C_i \wedge F$
12:    predicateWeightLearning ($newConj$, $CSST$)
13:    drop any predicate $C_{ij}$ frome $newConj$, if $C_{ij}.weight < \tau_p$
14:    if $F \neq \emptyset$ then
15:        for each conjunction $F_p$ in $F = F_1 \vee F_2 \vee \cdots \vee F_k$ do
16:            if $p == 1$ then $C_i = C_i \wedge F_1$
17:            else $newConjs[newConjsCunt++] = C_i \wedge F_p$
18: $Q' = Q$
19: for each new conjunction $C_{new\_i}$ in $newConjs$ do
20:     $Q' = Q' \vee C_{new\_i}$
21: compute_CSS_tables($Q'$, $FT$)
22: $Q'$ = assignConjunctionWeight ($Q'$)
23: drop any conjunction $C_i$ frome $Q'$, if $C_i.weight < \tau_c$
```

After all conjunctions are individually modified, their final conjunction weights need to be determined. Again, we use the conjunction's CSS table. Intuitively, non-conflicting positive cases in the CSS boost the importance of the conjunction in the query. Given a set of tuples in the CSS, we can measure how well the conjunction performed in the query after refinement by computing the ratio of non-conflicting positive cases captured by this conjunction and the total number of positive cases. We use this ratio as an overall conjunction weight measure. We specify our overall RAF query refinement algorithm in pseudo-code as follows:

The algorithm takes the previous query and the feedback table as input and generates a refined query. In the above algorithm, two things have been left unspecified (1) the algorithm to resolve the conflict set of a conjunction (line 10), and (2) the algorithm to learn the weight template for a given conjunction (line 12). As RAF is an open framework, different approaches can be used for the two learning tasks. Below, we provide algorithms we developed for this purpose.

3.3 Learning Predicate Weights of a Conjunction

Learning predicate weights corresponds to learning their relative importance in a conjunction. Since each dimension in the CSS corresponds to the similarity value of a predicate in a conjunction, we can map the weight learning problem to a classification problem over the CSS. Since we use the weighted summation model, we seek a hyper-plane in the CSS that separates the set of tuples marked deemed relevant, R, from the set of tuples deemed irrelevant, IR. We adapt the linear optimization process described in [8] for this purpose. The average complexity of this process is $O(n\times \mid p \mid)$, where n is the number of tuples with feedback and $|p|$ is the number of predicates in the conjunction.

Table 3. Conjunction Scores table example

ID	Loc		Sal	Dur	Desc		FB
	"SD"	**"SF"**			**"DB Dsger"**	**"DB Dvper"**	
2	0.8	0.4	70K	1	0.7	0.8	NOK
3	1	0.3	60K	1.5	1	0.7	OK
4	0.3	1	70K	1.5	0.7	1	OK

3.4 Learning New Structure of a Conjunction

The purpose of modifying a conjunction is to resolve conflicts present in the result set. Therefore, given a CSS table, we choose only the tuples that have conflicts, and use them in structural learning. For example, tuples 2, 3 and 4 in Figure 2 will be used in this learning. The original tuples in a `feedback table` form a learning set (*LSet*). Before we can apply our learning algorithm to this set, we need to convert it to a suitable format called a `scores table`. This conversion is needed because performing query refinement directly on the *LSet* is not always possible as database attributes can have complex and non-ordinal (numeric) attributes. The `scores table`, which consists of conflicting tuples, contains only ordinal values.

Data Preparation: Scores Table Generation.– This table is generated by retaining columns for ordinal attributes from the conflicting tuples in the *LSet* and converting complex and non-ordinal attributes into similarity measures. This similarity measure is computed by taking every value of each complex attribute in the *LSet* and calculating its similarity to all the other values of that attribute in the *LSet*. For example, tuples 2, 3 and 4 (i.e., the conflicting set) in Figure 2 form the conjunction scores table shown in Table 3. In this example, columns for ordinal attributes like Salary and Duration are identical to those in the `feedback table`. For the remaining non-ordinal attributes we created a column for each attribute–value pair. Each entry in these columns measures the similarity between the value in the heading and the attribute values from the tuples in the `feedback table`. For instance, the first entry under the column Location="SD" (i.e. 0.8) is the similarity score of the value "SD" and the location value of tuple 2 which is "LA". Since we want to learn predicates that focus on relevant tuples, only attribute values from records that are marked "Relevant" are used to from columns in the scores table. Consequently, no column is created for attribute values like "LA" because the tuple having this value is marked "Not Relevant".

Learning New Structures from a Scores Table.– Given a conjunction scores table, we can use any DNF learner (see section 5 for a review) to extract a set of hypotheses that explains/classifies the scores table. In this paper, we use a modified decision tree learner, namely C4.5 [15]. The original C4.5 employs a greedy divide-and-conquer strategy to build a decision tree. Given a labeled data such as a `scores table`, the algorithm initializes a decision tree with one leaf node that represents the whole data. It tests each attribute, and chooses the best attribute (A) and value (v) pair, which, if used to split the data into two portions – one with values in attribute $A \geq v$, another with values $< v$ – results in maximum entropy gain. It also records this split in the decision tree by splitting the original leaf nodes to two new leaf nodes; the old leaf node becomes the root of the two new leaf nodes. The algorithm recursively tests and splits the leaves until either all points in each partition belongs to one class (e.g., all marked with positive feedback or all marked with negative feedback), or it becomes statistically insignificant to split further. C4.5 then generates DNF rules/conjunctions from the decision tree.

Our main modification to C4.5 is an additional stopping condition. If we allow C4.5 to keep splitting its leaf nodes that represents a portion of data, the leaf nodes will get purer and purer towards a class. We instead stop splitting the node once a leaf node no longer has conflicts. Then, once the decision tree is constructed, a set of rules/conjunctions are derived from it. We, then, filter out the conjunctions that do not have any associated feedback tuples. The remaining conjunctions form the logical formula F used in the activation algorithm. Note that these conjunctions are also similarity based conjunctions since they are learned from the scores table. For example, from the `scores table`, table 3, the classifier generates a predicate $Dur \geq 1.5$, that removes the original conflicts.

To improve efficiency, we push similarity computations into the tree building process. This approach also avoids materialization of the `scores table`. Furthermore, we do not need to construct all `scores table` columns if we have already obtained a reasonable split point. The worst case complexity of this algorithm is $O(\mid R \mid \times d \times n \times (log(n))^2)$, in which $\mid R \mid$ is the number of relevant cases, d is the number of dimensions of the original feedback table, and n is the size of the feedback table. In practice, the number of the distinct relevant values of an attribute (e.g., Loc) is considerably smaller than $\mid R \mid$. Therefore, the complexity is close to standard decision tree complexity, which is $O(d \times n \times (log(n))^2)$.

4 Experiments

In this section, evaluate the effectiveness of RAF. We first present our experimental setup including the datasets we used and our synthetic query generator. We then evaluate RAF against four well-known algorithms.

4.1 Experimental Setup

We ran all our experiments using a P3-800MHZ PC with 256MB RAM. The similarity retrieval component is implemented using UDF features in IBM DB2. The refinement component is implemented as a stored procedure in IBM DB2. We use IBM OSL package [6] as our linear problem solver.

We used 12 datasets from the UCI machine learning repository [10], all of which have class labels. These datasets represent different domains of interests. For discrete attributes, we analyzed each pair of values of an attribute to decide the similarity value between them. For continuous attributes, we used the formula in section 2.1 to compute the similarity value of any two intervals.

To evaluate our query refinement method, we formulate two related queries, a *target* query q_t which simulates a user's real information need, and an *initial* query q_i which simulates his initial knowledge when starting the search. The query q_t can be more *general* or *specific* than q_i in terms of its DNF structure. We built a query generator that generates a set of query pairs $\langle q_1, q_2 \rangle$, where q_1 is more specific than q_2. If we set $q_t = q_1$ and $q_i = q_2$ we evaluate our algorithms on refining a general query to a specific query (i.e. G2S in our experiments). Conversely, if $q_t = q_2$ and $q_i = q_1$, we evaluate our algorithms on refining a specific query to a general query (i.e. S2G in our experiments). We generated 20 target and initial query pairs for each dataset getting a total of 240 pairs. Also, to simulate a realistic query, the size of results from q_t should be small compared to the database size. We only take top 20 records from q_t as its return set. The largest dataset we tried has nearly 32,000 records; this makes the target size less than 0.06% of the database size.

Algorithms Tested. We compare RAF with four existing methods. The first method, which we refer to as OCM (from the authors' last name initials) focuses on learning conjunctions [13]. OCM is similar to our approach in that it also refines the initial query structure using relevance feedback. The second method is FALCON [20], which does not refine the initial query explicitly, but uses the positive feedback to rank the data. The third method, Rocchio's method [16], is a standard vector based IR approach. In our experiment, we use attribute-value pairs as the vectors. Hence, Rocchio's method ignores the query structure and also similarity measures on the attributes. Since our refinement task essentially performs a classification based on the relevance feedback, one can directly apply an existing classification package if the data types can be mapped to the required types. This can be easily done for the 12 datasets we used, but not in general. Hence, the fourth method we compare our approach to is a well-known decision tree classifier C4.5 [15]. C4.5 takes the feedback data as a training set, and generates a classifier. It ignores the initial query structure, and predefined similarity measures on the data attributes. The refinement system uses this classifier to assign confidence levels as scores to the remaining data tuples.

Evaluation Process. The above algorithms are evaluated based on:

1. Efficiency: time taken to refine a query
2. Effectiveness: precision and recall measures
3. Simplicity of a refined query: number of the predicates in the query

To evaluate the effectiveness of a refinement algorithm, we simulate the desired user concept by first executing a target query and placing the top 20 tuples that satisfy the target query into a set R, the *relevant* records. Then, we execute the initial query (i.e., iteration 1). We compute the precision level at every 5% of recall interval (i.e. every relevant point retrieved) until all the relevant tuples are retrieved. To study effectiveness, we form the first learning set L by adding the top retrieved tuples containing exactly two relevant tuples from the initial query result. If a refinement algorithm performs well at a refinement iteration, it should have good precision level in all recall intervals. The refinement algorithm uses

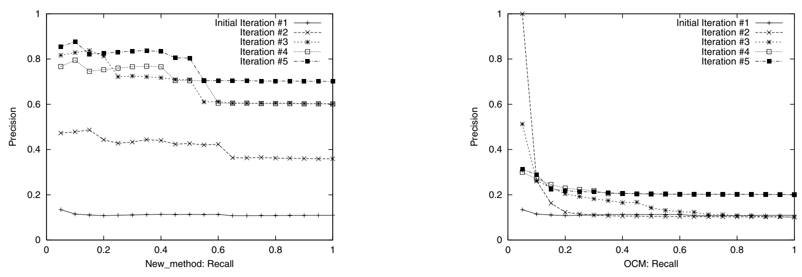


Fig. 3. Precision-Recall: RAF Vs. OCM

set L as the `feedback table` to learn a refined query. In subsequent iterations, the learning set L is updated by adding the top ranked tuples of each iteration that contain two new relevant tuples. Hence, L continues to grow accumulating all the relevance feedback gathered starting from the initial iteration. In our experiments, we execute the refinement algorithm four times (i.e., five iterations) to evaluate its performance in different iterations.

4.2 Results

As shown in table 4, average execution time for a refinement cycle in our algorithm averages around 0.5 seconds and ranges between 0.2 to 0.7 seconds. Falcon and Rocchio algorithm are not included in this table since they do not generate queries explicitly. RAF is consistently more efficient than OCM. This is because OCM always starts structure tuning which is typically more expensive than weight tuning. In RAF, the activation procedure does not choose structure tuning if there are no conflicts. RAF runs slower than C4.5. This is expected since the feature space (i.e., scores table) used in RAF is larger than the original data's feature space. Furthermore, RAF also performs weight tuning.

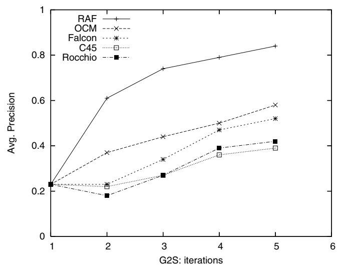
In Figure 3, we show the refinement effectiveness of RAF and OCM on the dataset *adult*, which is the largest dataset in our experiment. In each of the two graphs, each line represents average precision over 20 similarity queries at every 5% of recall interval. We use the average precision at each 5% recall interval as a measure of retrieval quality. If a method does well in an iteration, this average should be high. For example, average precision of RAF at iteration 1 is 11%, and after one refinement cycle it increases to 40%. OCM still remains at 11%. Hence, RAF outperforms OCM after the first refinement cycle.

Table 4. Average Refinement Time per Query (sec.)

G2S Avg. Refine Time(sec.)			S2G Avg. Refine Time(sec.)		
RAF	*OCM*	*C*4.5	*RAF*	*OCM*	*C*4.5
0.4	0.8	0.2	0.5	0.8	0.3

Table 5. Avg. Number of Predicates in Query

Dataset	G2S				S2G	
	Init	*It*2	*It*5	*Target*	*It*2	*It*5
RAF	4.8	5.1	5.6	6.7	6.9	5.1
OCM	4.8	15.1	15.9	6.7	15.3	9.7
Rocchio	4.8	69.6	76.3	6.7	76.3	97.9
C4.5	4.8	1.1	1.8	6.7	0.8	1.5



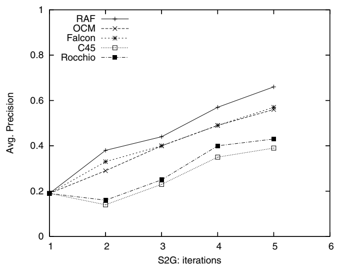


Fig. 4. Precision Average at Each Iteration

To further evaluate the learning behaviors of each algorithm in different iterations, Figure 4 plots the overall average over different iterations. We observe that RAF has very good learning behaviors in both learning scenarios (i.e., G2S and S2G), such that it has much better precision at each iteration. We also notice that without learning, the initial query performs badly in general (i.e., iteration 1). Although OCM attempts to improve the retrieval quality by modifying the initial query concept, it performs worse than RAF. The main reason is that RAF focuses directly on resolving the essential feedback conflicts, but OCM simply tries to fit concepts (i.e., predicates) that match to the feedback. Also, for some cases OCM may wrongly add a set of predicates while simple weight modification can achieve the correct ranking. RAF outperforms OCM on average by 24% in terms of average precision after first refinement cycle for G2S learning. This average is computed over the 240 query pairs. RAF and OCM outperform the other three methods in both $G2S$ and $S2G$ learning scenarios. This clearly shows the merit of considering the initial query structure during the refinement process. C4.5 and Rocchio perform the worst since they ignore the initial query structure and the predefined attribute level similarity functions during refinement cycles.

In addition to precision and recall measures, another interesting measure of a refinement algorithm is the simplicity/complexity of its refined queries. Table 5 shows the average number of predicates over the 240 queries at different iterations. The initial queries for G2S case contain 4.8 predicates on average, and target queries have 6.7. For the S2G case, the number is reversed. As we observed, the number of average predicates in RAF at different iterations lies in between the average number of predicates in initial and target queries. This increases query understandability since the refined query is not far away from the target query. OCM has about twice as many predicates on average as the target. The main reason of this is that OCM cannot represent disjunctive concepts explicitly; it uses additional predicates to simulate disjuncts. We also report the average number of vectors used by the Rocchio's method at different iterations, and the number of vectors used could be 10 times bigger than the number of target predicates. C4.5 is at another extreme, which, due to the small number of positive feedback available at each iteration, generates very small number of rules with only one or two conditions. Falcon is not included because it does not produce queries.

5 Related Work

The work most related to this paper is [13], which proposed a query refinement framework on top of ORDBMSs for learning SQL queries from user interactions. There are two major limitations in [13]. Firstly, it did not consider weight and structure learning separately, and at each iteration both weight and structure were modified while modification of only one of the two would have sufficed. This resulted in, at times, an over aggressive policy that adds/drops wrong predicates from the query. Secondly, [13] considered only learning a limited set of conjunctive queries; it did not support learning disjuncts. If the optimal query the user had in mind and the feedback was consistent with a disjunctive query (e.g., *Salary* $>$ *65K OR Duration* $>$ *2 years*), the approach would attempt to simulate a disjunction via a conjunction. That is, it would attempt to learn the query of *Salary* $>$ *65K AND Duration* $>$ *2 years*. This results in a suboptimal query with poor retrieval performance and poor interpretability.

Query refinement via feedback has been explored extensively in the context of text document retrieval in the information retrieval literature [16, 1]. Generally, a vector space model is assumed (i.e., Rocchios method [16]). Refinement tasks focus on how to weight the elements in the vector space. There is no explicit query formulation and attribute similarity measures. IR models have also been generalized to multimedia domain. Query refinement techniques have also been exploited in the multimedia domains, e.g., MARS [17, 9], Mindreader [7], FALCON [20]. FALCON generalizes the refinement model to any suitably defined metric distance function. As long as the distance between two tuples can be properly defined, FALCON can be applied. It uses the relevant examples as the query, and rank the database based on the aggregate distance measure. However, since FALCON does not consider the original query formulation, it performs poorly when the relevant set is very small.

Although not from the point of view of database queries, there is a considerable body of work on learning DNF and CNF formulas from examples. DNF learners in the literature can be grouped into two as divide-and-conquer based (e.g. decision tree learners [15]) and separate-and-conquer based or covering algorithms. Among the large number of algorithms in the latter category, the AQ family of algorithms [2], CN2 [3] and PFOIL [11] are popular. In these approaches, predicates have no similarity semantics (i.e., are crisp conditions). It is also unclear how to incorporate initial queries and similarity functions into these approaches. Furthermore, these algorithms normally require large amount of input (i.e., training and testing ratio) before deriving good hypothesis.

6 Conclusions

In many search environments, a user normally has imprecise specification of what he wants. We provide a system to enable users express imprecise queries, and refine it interactively by supplying relevance feedback. We identified two distinct tasks of similarity query refinement: refining query structure and determining the relative importance of predicates and conjunctions. We proposed a set of novel algorithms to control weight and structure learning. We implemented these algorithms and the extensive experiments we conducted showed that RAF consistently outperforms previously suggested techniques both in terms of retrieval quality and query simplicity.

References

1. Ricardo Baeza-Yates and Ribeiro-Neto. *Modern Information Retrieval.* ACM Press Series/Addison Wesley, New York, May 1999.
2. E. Bloedorn, R. S. Michalski, and J. Wnek. Multistrategy constructive induction: AQ17-MCI. In *Proc. of the 2nd Int. Workshop on Multistrategy Learning*, pages 188–203, 1993.
3. P. Clark and T. Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3(4):261–283, 1989.
4. Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *Proc. of the 15th ACM Symp. on PODS*, 1996.
5. Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
6. IBM. IBM linear optimization package: http://www-3.ibm.com/software/data/bi/osl/pubs/lpsol/lpuser.htm.
7. Y. Ishikawa, R. Subramanya, and C. Faloutsos. Mindreader: Querying databases through multiple examples. In *VLDB*, 1998.
8. O.L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis. In *SIAM*. 1990.
9. S. Mehrotra, Y. Rui, M. Ortega, and T. Huang. Supporting content-based queries over images in mars. *Proc. of IEEE-ICMCS97*, 1997.
10. C. J. Merz and P. Murphy. UCI Repository of Machine Learning Databases. http://www.cs.uci.edu/ mlearn/MLRepository.html, 1996.
11. Raymond J. Mooney. Encouraging Experimental Results on learning CNF. *Machine Learning*, 19(1):79–92, 1995.
12. M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, , and T. Huang. Supporting ranked boolean similarity queries in mars. *IEEE Trans. on Data Engineering*, 10(6), December 1998.
13. Michael Ortega-Binderberger, Kaushik Chakrabarti, and Sharad Mehrotra. An Approach to Integrating Query Refinement in SQL. In *Proc. EDBT*, March 2002.
14. Kriengkrai Porkaew, Sharad Mehrotra, Michael Ortega, and Kaushik Chakrabarti. Similarity search using multiple examples in mars. In *Proc. Visual'99*, June 1999.
15. R. Quinlan. *C4.5: Program for Machine Learning.* Morgan Kaufmann, 1992.
16. J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The SMART Retrieval System: Experiments in Automatic Document Processing*, pages 313–323. Prentice Hall, 1971.
17. Y. Rui, T. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. *IEEE Proc. of Int. Conf. on Image Processing*, 1997.
18. Y. Rui, T. Huang, M. Ortega, and S. Mehrotra. Relevance feedback: A power tool for interactive content-based image retrieval. *IEEE Trans. Circuits and Systems for Video Technology*, 1998.
19. G. Salton. The use of extended boolean logic in information retrieval. In *SIGMOD*. 1984.
20. L. Wu, C. Faloutsos, K. Sycara, and T. Payne. FALCON: Feedback adaptive loop for content-based retrieval. *VLDB*, 2000.