A Framework for Refining Similarity Queries Using Learning Techniques

Yiming Ma, Qi Zhong, Sharad Mehrotra, Dawit Yimam Seid
Information and Computer Science
University of California, Irvine
{ maym, qzhong, sharad, dseid }@ics.uci.edu

ABSTRACT

In numerous applications that deal with similarity search, a user may not have an exact idea of his information need and/or may not be able to construct 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 the feedback on the relevance of the retrieved objects to refine the query. In this paper, we explore such a refinement strategy for a general class of SQL similarity queries. Our approach casts the refinement problem as that of learning concepts using examples. This is achieved by viewing the tuples on which a user provides feedback as a labeled training set for a learner. Under this setup, SQL query refinement consists of two learning tasks, namely learning the structure of the SQL query and learning the relative importance of the query components. The paper develops appropriate machine learning approaches suitable for the two learning tasks. The primary contribution of the paper is a general refinement framework that decides when each learner is invoked in order to quickly learn the user query. Experimental analysis over many real life datasets and queries shows that our strategy outperforms the existing approaches significantly in terms of retrieval accuracy and query simplicity.

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 increasingly used for exploratory data analysis and retrieval where a user may not initially have a clear mental model of his exact information need [27].

Although similarity retrieval can be supported using extensibility features (UDFs and extended data types) offered by current object-relational systems, many research challenges remain. One such challenge, that has received a significant research attention recently, is that of efficient evaluation of similarity search. Many query processing techniques to optimize progressive retrieval of top-k matching answers have been devised (e.g., [7, 19, 9]).

The other significant challenge (which is the focus of this paper) is that of mechanisms to formulate and refine similarity queries. As mentioned above, in applications requiring similarity search (e.g., catalog searches, exploratory data analysis), users may not be sure of their specific information need at the beginning of the search. Such a user may only be able to provide a rough approximation of the desired query. Tools that overcome the "initial query" problem by interpreting user's information need from his interactions with the data will greatly benefit the search experience. Another motivation for such tools is the subjective interpretation of the meaning of similarity: different users may attach different levels of importance to the various search criteria of the query. For example, in a job seeker's search query, "within 30 miles of Los Angeles" may be less important to a user than "salary greater than to $65K". In contrast, the criteria of vicinity of Los Angeles might be very important to another user. Ideally, the system should be able to automatically tune the meaning of similarity based on user interactions.

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 criterias, 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 in the context of similarity search has been explored extensively in the context of text document retrieval in the information retrieval literature [24, 27, 4, 2]. More recently, its effectiveness in feature-based image and multimedia similarity retrieval [21, 26, 11] as well as similarity search over metric spaces [28] has also been established. In work [20], authors considered the problem of refining SQL queries from user interactions in an object relational database. The paper explored a limited set of SQL queries and illustrated that even simple techniques (i.e., mostly extensions of refinement approaches studied in the IR literature) can significantly enhance the users' search experience.

This paper considers the problem of refining a general class of SQL queries. In contrast to the ad-hoc approaches in [20], 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 [22]) to learn a new query representation that will replace the original query. Such a strategy to "learn" a query from feedback does not perform well in practice. Reasons for this include that classifiers do not work well when the training set is very small as is the case in our setting when a user may provide feedback on only a few records. Furthermore, the naive strategy largely ignores the initial query provided by the user; it may get trapped in a wrong hypothesis simply since the hypothesis fits the few examples that the user provided input on. In addition, it is very difficult to incorporate user defined types and similarity functions into these models. Consistency is also an important issue in the refinement task. Many existing classifiers are sensitive to the inputs. The models built from different iterations could differ substantially. This also makes these strategies less usable in refinement context.

In our approach, we consider the 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. The 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 and depends upon the system for query formulation. 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. A key issue in refining a query is to determine when to invoke the structure/weight learner. We develop a formal basis for making such a decision based on which an activation procedure is designed.

The major contributions of this paper are summarized as follows:

(1) We provide a powerful framework for refining a general class of SQL similarity queries that combines a multimodal refinement activation procedure with machine learning techniques to adjust both query predicate weights and query structures.

(2) We suggest 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 the 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

This section introduces our interpretation of similarity search conditions and corresponding refinement.

2.1 Similarity Search Conditions

We assume an analyst expresses his hypothesis as a similarity search condition which takes the form of DNF Boolean expressions over similarity predicates. That is, we organize the query as a disjunction of conjunctions of similarity predicates. Formally a query $Q = C_1 \vee C_2 \vee \ldots \vee C_n$ be a DNF expression where $C_i = C_{i1} \wedge C_{i2} \ldots, C_{in}$ is a conjunction, and each C_{ij} is a predicate. Note 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 The DNF formula helps us in structural learning in the refinement framework as will become evident later.

Similarity search differs from traditional exact-match search in that it generates domain-dependent scores that indicate how closely a property of an object (tuple) matches some target values (the similarity search condition). A *ranking function* then aggregates the scores from individual similarity predicates according to the structure of the DNF search condition and a corresponding set of weights to indicate the importance of each similarity predicate. The resulting answers are presented to the analyst in decreasing similarity score order.

Similarity Predicate: is defined over a domain of a given data type. It is a function with two inputs: (1) an input attribute value from a data tuple (t), (2) a target value which can be a point or a range. It returns a similarity *score* in the range [0,1]. The function has the form:
$Similarity\ Predicate(input\ value, target\ value) \rightarrow score$

We complement the similarity condition (in DNF form) with a *template of weights* that matches the structure of the condition and associates a weight to each predicate in a conjunction and also to the conjunction in the overall disjunction.

DNF Ranking Function is a domain-specific function. It uses the weight template to aggregate the scores for each tuple. It 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 each predicate of 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, in which a query is typically expressed as a single conjunction [24, 27, 4, 2]. For an overall score of the query (disjunction), we use MAX on the weighted conjunction scores. The MAX function is one of the most popular aggregation functions on a disjunction [7] [1].

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 in terms of retrieving relevant tuples. For example, the importance can be measured as percentage of relevant tuples covered by this conjunction.

We generically apply this template to aggregate scores through a function we call $RankVal$.

Definition 2.1 (RankVal) For a given similarity query S and a tuple t, a $RankVal$ function takes a weight template and individual similarity scores for each similarity predicate in S as inputs, aggregates the similarity scores and returns the overall score of t.

The aggregated tuple level scores ranked by $RankVal$ are used for ranking the tuples. The final score of a tuple is in the range of [0, 1], a similarity query will return a set of records with $score \geq \alpha$ (also called α *cut* or threshold), and this set of tuples forms the return set of a similarity query.

Table 1 shows an example data table which we will use throughout the rest of the paper to illustrate our approach.

[1] Many other alternative ranking functions can beused instead. For example, [19] explores numerous options including a probabilistic and fuzzy interpretation of similarity.

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 1: Example data table

It has four attributes pertaining to a contract database job application: location, salary, duration (Employment), and Job description. In practice, many more attributes may be presented. A typical task is to find jobs that conform to a certain desirable hypothesis. The hypothesis is expressed as a DNF search condition:

Example 2.1 (Similarity search condition) An example search condition on Table 1 is:[2]

(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 RMDB that supports UDF. Here is an example of implementing the above similarity search condition using a nested SQL statement.

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, (**SELECT** ID,
LocNear (Loc, "SN") **AS** l_s,
DurClose(Dur, 2) **AS** D_{s1},
SalGreater (Sal, 65000) **AS** S_s,
FROM Job) **AS** Score
WHERE Job.ID=Score.ID **AND** $S \geq \alpha$
ORDER BY S **DESC**

A corresponding weight template may be:
$(W_1(w_{11}, w_{12}), W_1(w_{21}, w_{22})) = (0.9(0.4, 0.6), 0.8(0.4, 0.6))$. The $RankVal$ function uses this template to aggregate similarity scores as: $MAX[0.9 \times (0.4 \times l_s + 0.6 \times D_{s1}), 0.8 \times (0.4 \times S_s + 0.6 \times D_{s1})]$.

This search finds jobs located near SN or that pay more than 65K and have a duration close to 2 years. The search has 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. They can be implemented by database developers as user-defined functions.

Here the user is more interested in a record that has duration close to 2 years than one that is close to SN or pays more than 65K as the weight assigned to the $DurClose$ predicate is higher. Although tuples with ID 1 and 4 receive same conjunction scores from the first and second conjunction, the first tuple is ranked higher since the conjunction weight for the first conjunction is higher. Therefore, the tuple with ID 1 should be returned as the top record.

2.2 Similarity Condition Refinement

The exploratory nature of similarity search makes it difficult for an analyst to properly choose the predicates and weights that yield the best results. We refine similarity conditions using *relevance feedback* to adapt the predicates, condition structure and corresponding weights to the subjective needs of the analyst. Given a search condition q, a set r of top-k records returned, and relevance feedback f on these records (i.e., a triple $\langle q, r, f \rangle$), the refinement problem is to modify the search condition q in such a way that, when re-executed, it will return more relevant records. For example, while the condition given in Example 2.1 may return the best results that satisfy a user's information need, he may have started searching with a different query:

Example 2.2 (Initial search condition) Without a clear idea of his exact information need, he may start with the search that approximately captures his information needs with the following similarity search condition:

$((LocNear(loc, "SN")$ **AND** $DescClose(Desc, "DBA"))$

With a corresponding weight template of: $(1(0.5, 0.5))$. This example also illustrates that user may not be aware of the data stored in the database in the beginning, and therefore he does not know how to formulate the exact query that returns the data that most interest him. For example, he uses *Description* similar to "DBA" as one of the search condition, but after seeing a few records with description same as "DBA", he may be unsatisfied with other attributes associated with these tuples. He may change his interests away from this search condition. Hence, the condition becomes irrelevant to him in his final query.

In this paper, we deal with the problem of refining search conditions like the above example through the use of relevance feedback. Postulated as above, the refinement involves two operations: (1) updating the structure of the search condition, and (2) updating the template of weights. These two operations re-order the records and hence provide different result set to a user. We update the structure through addition and/or deletion of predicates. We update the weight template that captures the relative weights of predicates by determining the extent to which each predicate contributes to the retrieval of tuples that match users' needs.

3. LEARNING QUERIES FROM USER FEEDBACK

3.1 Problem formulation

The problem of query refinement can be cast as a problem of learning a classifier if we view the records with relevance feedback as a set of training examples for a classifier. Hence, theoretically any concept learning method can be employed for query refinement provided that it can perform well on a small number of examples.

However, it is important to note that 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 paper, we consider two different types of learning algorithms (classifiers). One focuses on the query weight tuning, and another focuses on the query structure tuning. We introduce an activation algorithm, which controls the overall learning process.

[2]Note that this search is different from the one used in [20] as it extends that approach by supporting disjunctive terms. Hence this formalism is more powerful.

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

Table 2: Example feedback table

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

Table 3: Example CSS table

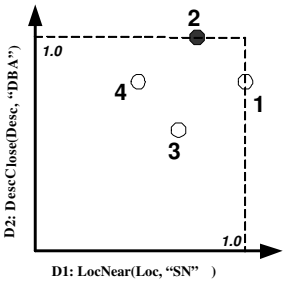

Figure 1: Example CSS in Table 3

3.2 Basic Tables

For each query session we maintain the following two types of tables:

(1) An *Answer Table* contains resulting tuples (that are presented in ranked order) as well as the similarity scores assigned to each.

(2) A *Feedback Table* stores the relevance feedback given by the user on tuples in the *answer table.* [3] Table 2 shows an example of such a table.

3.3 Refinement Framework

We introduce a general framework to control the refinement process that consists of two interrelated learning tasks – updating the weight template and modifying the structure of the query. The key aspect of the framework is an activation procedure that, given the user feedback, decides which 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 the rank order, structural changes can dramatically modify the return set by bringing objects deemed irrelevant to the original query to the top of the retrieved set. For this reason, structural changes to the query must be made conservatively since an incorrect change may lead a refinement along a wrong path causing the result set to become worse instead of better. The activation procedure we develop in this paper invokes structural modifications only when feedback provided by the user requires that such modifications be made and the desired effect cannot be achieved by reweighting the different query components. Identifying situations in which structural modifications are needed requires us to explore the limitations of weight learning in the context of refinement.

We next develop some notation that will allow us to formally explain our idea. Let us 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. Therefore, a tuple can be transformed as a point in the CSS. For example, CSS Table 3 shows an example CSS for the tuples in the feedback table (i.e., Table 2). We can also visualize this space in Figure 1. For the conjunction similarity space corresponding the conjunction, we define a notion of CSS domination and CSS conflict below.

[3]We consider binary relevance judgments for feedback values: tuples are either "Relevant" or "Not Relevant." Finer grained distinctions are possible, but we focus on the above for simplicity reasons.

Definition 3.1 (CSS domination) For a given conjunction similarity space (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 1, tuple 2 dominates tuple 3 and 4, but does not dominate tuple 1. Similarly, tuple 1 dominates tuple 3 and 4, but not tuple 2.

Definition 3.2 (CSS conflict) For a given CSS, a pair of feedback tuples $(t1, t2)$ conflict to each other if $t1$ receives negative feedback, $t2$ receives positive feedback, and $t1$ dominates $t2$. In addition, if $t1$ and $t2$ have equal values in all dimensions, it is also a conflict.

The conflicts in the CSS space in Figure 1 are between tuple 2 and 3, also tuple 2 and 4, but not tuple 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. As a result, the query does not capture the user's information need and must be modified. Unfortunately, simply modifying weights associated with the predicates would not resolve the conflict as is stated in the following lemma.

Lemma 3.1 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 [8], 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. Consider a conflicting tuple pair $(t1, t2)$, 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 (i.e., $t1$ and $t2$ have equal values in all dimensions) in CSS conflict definition (i.e., Definition 3.2), Lemma 3.1 is still true. By adding the condition, we potentially capture more cases that could not be resolved by weight adjustments. □

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. The activation procedure during refinement attaches feedback to different conjunctions by assigning each feedback to the highest scored

conjunction. [4] For each conjunction C_i, it then determines if the assigned feedback contains any conflicts with respect to the conjunctions C_i. If a conflict set is identified, structural modification to the query is invoked based on the conflicting set. For example, in Figure 1 (i.e., empty circle represents positive feedback), tuple 2 conflicts with tuple 3 and 4. To resolve the conflicts, the activation procedure invokes the structural learning using tuples 2, 3, and 4. Specifically, the structure learning algorithm will attempt to learn a predicate that when added to the conjunction will resolve the conflict. In the current example, the structural learning algorithm may suggest the addition of a predicate $Duration \geq 1.5$ to the conjunction. In general, depending upon the mechanism used, the conjunction may be augmented by not simply a single predicate but a logical formula F comprising of multiple predicates connected via logical connectives. In the new formula (i.e., $newConj = C_i \wedge F$), most of (or all) the conflicts associated with the original conjunction have been resolved. We temporarily treat F as a pseudo predicate, so that the new formula becomes a conjunction and the weights of predicates in this conjunction are rebalanced using a predicate weight learning approach. If a weight of a particular predicate falls below a threshold, the predicate is dropped. We assume the assigned weight of F is W_F.

In order to combine the conjunction (i.e., $newConj$) into our DNF-formed query, conversions on the pseudo predicate F are required. Assume $F = F_1 \vee F_2 \vee \cdots \vee F_k$ is itself in DNF. The predicates in C_i can be distributed over the F_p, resulting in a new candidate set of conjunctions $(C_i \wedge F_1) \vee (C_i \wedge F_2) \vee \cdots \vee (C_i \wedge F_k)$. 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 (i.e., C_i) weight to initialize the weights of candidate conjunctions (i.e., $C_i \vee F_p$).

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 the non-conflicting positive cases captured by this conjunction over the total number of positive cases. We use this ratio as an overall conjunction weight measure. We specify our overall query refinement algorithm in pseudo-code below:

ActivationAlgo()
Input: Query (Q), FeedbackTable (FT),
Output: New Query (Q')
1. compute_CSS_tables(Q, FT)
2. $newConjunctions = \emptyset$
3. $newConjunctionCount = 0$
4. Foreach conjunction C_i in a query Q
5. $CSST$ = get_CSS_table(C_i)
6. $Conflicts$= computeConflictSet($CSST$)
7. $F = \emptyset$
8. If $|Conflicts| \geq 1$
9. F= learnStructure(C_i, $Conflicts$)
10. endIf
11. $newConj$=$C_i \wedge F$
12. predicateWeightLearning ($newConj$, $CSST$)
13. Foreach predicate C_{ij} in $newConj$
14. If $C_{ij}.weight < \tau$
15. $newConj$.dropConjunct(j)
16. endIf
17. endFor
18. If $(F \neq \emptyset)$
19. Foreach conjunction F_p in $F = F_1 \vee F_2 \vee \cdots \vee F_k$
20. If $(k == 1)$
21. $C_i = C_i \wedge F_1$
22. Else
23. $newConjunctions[newConjunctionCount++] = C_i \wedge F_p$
24. endIf
25. endFor
26. endIf
27. endFor
28. $Q' = Q$
29. Foreach conjunction C_i in $newConjunctions$
30. $Q' = Q' \vee C_i$
31. endFor
32. compute_CSS_tables(Q', FT)
32 Q' = assignConjunctionWeight (Q')
33. Foreach conjunction C_i in Q'
34. If $C_i < \tau$
35. Q'.dropConjunction(i)
36. endIf
37. endFor

The algorithm takes as input the previous query and the feedback table to generate a refined query. In the above algorithm, two things have been left unspecified (1) the algorithm to resolve the conflicting set for a conjunction (line 9), and (2) algorithm to learn the weight template for a given conjunction (line 12). In fact, it can be easily shown that the overall complexity of the activation procedure is dominated by these two algorithms. Many different approaches can be used for this purpose and integrated into the refinement activation framework described above. We provide specific algorithms we have developed for this purpose in the following two subsections.

3.4 Learning Predicate Weights of a Similarity Conjunction

Learning the predicate weights corresponds to learning their relative importance in a conjunction. Intuitively, the smaller the weight for a predicate, the less important the corresponding predicate. Since each dimension in the CSS corresponds to the similarity value of a predicate in the conjunction, we can cast the weight learning problem to a classification problem over the CSS. Since we use weighted summation model, we seek a hyper-plane that separates the relevant tuple set R (i.e., those that receive a positive feedback) from irrelevant tuple set IR (i.e., those that receive a negative feedback). With minor modifications to the solution proposed in [13], the linear programming solution is able to produce good weight configuration. In CSS, each point corresponds to a row vector (p_t) formed by the feedback tuple's predicate scores. Therefore, we wish that for a fixed threshold β, every positive feedback tuple $t+$'s weighted summation score is above β, that is:

$$Score(t+) = \sum_{i=1}^{d} w_i * p_{t+}(i) \geq \beta$$

Furthermore, for every negative feedback tuple $t-$, that received a negative feedback, the score is less than threshold (i.e., $Score(t-) < \beta$). In other word, the weight learner seeks a hyper plane ($p_t \cdot W^T = \beta$) that best separates the two sets in the CSS. To ensure the quality of this hyper plane, we impose following optimization requirements:

(1) When set R and IR are linearly separable, we would like the hyper plane maximize the separation margin between

[4] Since the aggregation function used on a disjunction is MAX, assign a feedback tuple to the highest scored conjunction is appropriate. For other aggregation functions, different assignment schemes should be used.

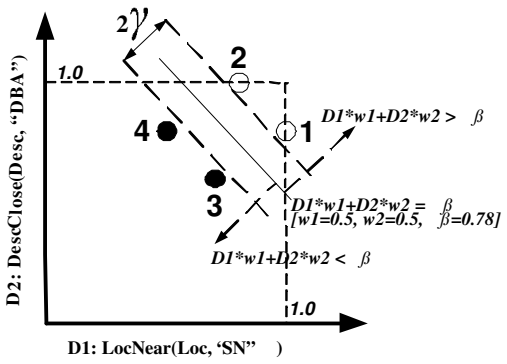

Figure 2: Learning weight template

two sets [5].

(2) When set R and IR are linearly inseparable, we would like to find the hyper plane reasonably good in separating the two sets.

The first requirement ensures an optimal separation hyper plane in the linearly separable situation. For example, Figure 2 shows a linearly separable case (i.e., tuple 1 and 2 are relevant, and tuple 3 and 4 are irrelevant). γ is the separation margin. The hyper plane nicely separates the two sets when γ reaches its maximum value.

The second requirement is important in determining CSS conflicts under WSM in Definition 3.2. It is obvious that conflicts can only happen when set R and IR are linearly inseparable. Therefore, if we can find a reasonably good hyper plane, we can use it to test the conflicts.

In Section 3.5 and Section 3.6, we show that we can cast the weight learning problem to a suitable optimization problem, and by solving the optimization problem, we are able to satisfy the requirements listed above.

3.5 Casting Weight Learning as an Optimization Problem

We can cast this weight learning problem to an optimization problem as in [13]:

$$Minimize: f(W,\beta,\gamma) = (\frac{Error(R)}{|R|} + \frac{Error(IR)}{|IR|}) \quad (1)$$

$|R|$ and $|IR|$ are the cardinalities of the set R and IR. The goal of the equation is to minimize the average classification errors defined by Equation 2 and Equation 3. The intuition of these two error functions is that when a point falls into its correct half-space, there is no error. Otherwise, the distance to the hyper plane is used as an error measure. γ is used to setup a separation margin.

$$Error(R) = \sum_{i=1}^{|R|} \max(-Score(t{+}_i) + \beta + \gamma, 0) \quad (2)$$

$$Error(IR) = \sum_{i=1}^{|IR|} \max(Score(t{-}_i) - \beta + \gamma, 0) \quad (3)$$

From [13], we know in the linearly separable cases (e.g., in Figure 2), the minimum value of the objective function is always zero with some positive γ values, and the hyper plane obtained can perfectly separate the two sets.

In the linearly inseparable cases, the meaning of the formulation is to minimize the average errors. We consider a hyper plane is of good quality if it minimizes the average errors. We further convert this formulation to a linear programming (LP) formulation in the next subsection. By imposing the right constraints, we are able to satisfy the two requirements in the beginning of this section.

3.6 Solving the Optimization Problem Using Linear Programming

Motivated by [13], we rewrite the optimization problem in Equation 1 as following LP formulation:

$$Minimize: f(W,\beta,Y,Z,\gamma) = \lambda\{g(W,\beta,Y,Z,\gamma)\} - \gamma \quad (4)$$

$$Where: \quad g(W,\beta,Y,Z,\gamma) = \frac{\sum_{i=1}^{|R|} y_i}{|R|} + \frac{\sum_{i=1}^{|IR|} z_i}{|IR|} \quad (5)$$

Subject to:

$$y_i \geq -Score(t{+}_i) + \beta + \gamma \quad (6)$$

$$z_i \geq Score(t{-}_i) - \beta + \gamma \quad (7)$$

$$y_i > 0, z_j > 0 \quad (8)$$

$$\sum_{i=1}^{n} w_i = 1 \quad (9)$$

$$w_i \in [0,1], \beta \in [0,1], \gamma \in [0, \sqrt{d}] \quad (10)$$

In this formulation, λ is a big constant. The constraints in Equation 6, Equation 7 and Equation 8 are from linear formulation proposed by [13]. The constraints in Equation 9 and Equation 10 are specially designed for the weight learning. The constraint in Equation 9 ensures a non-zero weight vector. The constraints in Equation 10 bound the limits of the target variables. In [13], there are no constraints on W and β, and γ is a constant that always equals to one. In our case, we need to bound the weight vector W and threshold β according to CSS definition. We also set γ as one of the target variables, and bound the γ by the maximum range (i.e., $[0,\sqrt{d}]$, where d is number of dimensions in CSS).

This formulation has two components. The first one (i.e., function $g(W,\beta,Y,Z,\gamma)$ focuses on the minimizing the misclassification errors, and the second one focuses on maximizing the separation margin. In fact, the function $g(W,\beta,Y,Z,\gamma)$ is originated from [13] as the objective function to minimize. Since λ acts as a big weight to the first component, the intuition is that optimization goal always focuses on minimizing errors, and only when errors are minimized, it then maximizes the separation margin (i.e., γ). Therefore, in the linearly separable case, the hyper plane is always able to perfectly separate the two sets, and have maximum margin. If we apply the formulation to the CSS shown in Figure 2 (i.e., a linearly separable case), we have a hyper plane with maximum separation margin (i.e., γ=1.2). In the linearly inseparable cases, the LP formulation is still able to return a good separation plane since it minimizes the misclassification errors.

We now formally show that the LP formulation can fulfill our weight learning requirements.

Lemma 3.2 Given λ approaches $+\infty$, if Equation 4 reaches minimum value (i.e., f_{min}) with a set of parameters P_0 $=\{W_0, \beta_0, Y_0, Z_0, \gamma_0\}$, then

[5] By maximizing the separation margin, the hyper plane obtained will have good generalization behavior.

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

Table 4: Scores table example

(a) $g(P_0) = g_{min}$

(b) γ_0 is the maximum value under condition (a)

PROOF. We prove it by contradiction. Assume the conclusion is incorrect, such that either (a) or (b) is incorrect.

- If (a) is incorrect, then $g(P_0) > g_{min}$. There must exist another set of parameters $P_1 = \{W_1, \beta_1, Y_1, Z_1, \gamma_1\}$, such that $g(P_1) < g(P_0)$.
  Because $\lambda \rightarrow +\infty$ and γ is bounded in $[0, \sqrt{d}]$, we have:
  $\lambda\ (g(P_0) - g(P_1) > \gamma_0 - \gamma_1$
  $\Rightarrow (\lambda\ g(P_0) - \gamma_0) > (\lambda\ g(P_1) - \gamma_1) \Rightarrow f_{min} > f(P_1)$, which contradicts to the claim that f_{min} is the minimum value of the objective function.
- If (b) is incorrect, then there exists $\gamma_2 > \gamma_0$ and set of parameters $P_2 = \{W_0, \beta_0, Y_0, Z_0, \gamma_2\}$, such that $g(P_2) = g_{min}$. $\Rightarrow (\lambda\ g(P_2) - \gamma_2) < (\lambda\ g(P_0) - \gamma_0) = f_{min}$, which contradicts to the claim that f_{min} is the minimum value of the objective function.

Based on the previous two points, we conclude Lemma 3.2 must be valid. □

From Lemma 3.2, we know if we minimize the objective function in Equation 4, we solve the optimization problem posed in Equation 1 with maximum separation margin γ.

In most practical cases, the average complexity of this optimization process is $O(n \times \mid p \mid)$, where n is number of feedback and $\mid p \mid$ is number of predicates in the conjunction.

3.7 Learning new Structure of a Similarity Conjunction

Our objective in modifying a conjunction is to resolve conflicts present in the result set for the conjunction. As a result, given a CSS table, we choose only the tuples that have conflicts, and use them in the structural learning. For example, tuple 2, 3 and 4 in Figure 1 are used in this learning. These tuples from *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, where we refer to as the scores table.

3.7.1 Data Preparation: The Scores Table Generation

Conversion to the *scores* table is necessary for query structure learning because our query refinement methods are based on numeric (ordinal) values. Since database attributes can have complex and non-ordinal attributes, performing query refinement directly on the *LSet* is not always possible. Hence, we generate the *scores* table which corresponds to conflicting tuples, and contains only ordinal values. This table is generated by maintaining 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, tuple 2, 3 and 4 (i.e., conflicting set) in Figure 1 form the conjunction scores table (i.e., Table 4). In this example, columns for ordinal attributes like Salary and Duration are exactly the same as 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 is a similarity value 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 value of the value "LA" and the location value of tuple 3 which is "SD". We only take the attribute values from records that are marked "Relevant" in the feedback table. This makes sense since we want to learn predicates that focus on relevant tuples. Hence, no column is created for attributes like "LA" because the tuple having this value is marked "Not Relevant".

3.7.2 Learning New Structures from a Scores Table

Given a conjunction scores table, we can use any DNF learner (i.e., the ones discussed in Section 5) to extract a set of hypothesis that explains/classifies the scores table. In this paper, we use a modified decision tree (i.e., C4.5 [23]) to resolve conflicts of a conjunction. The original C4.5 adopts a greedy divide-and-conquer strategy in building a decision tree for a classification purpose. 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 (i.e., into two portions, one with values in attribute $A \geq v$, another has 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 (i.e., 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 the partitions belong 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.

The main modification to the C4.5 we made to use it to resolve conflicts of a conjunctions is to add an additional stopping condition. If we allow decision tree construction algorithm like 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. The modification we made is that once a leaf node no longer has conflicts, we can stop splitting the node. When we finish constructing this decision tree, based on this tree, we can generate a set of rules/conjunctions, and filter out the conjunctions that do not have any relevant feedback tuples. The remaining conjunctions forms the logical formula F used in the activation algorithm. Note that these conjunctions are also similarity based conjunctions since they are learned from scores table. For example, if we have the scores table as Table 4, decision tree generates a predicate $Dur \geq 1.5$, by adding in this predicate, it removes the original conflicts.

One advantage of choosing C4.5 is that all data points assigned to a conjunction are disjoint from other conjunctions. Therefore, if a conflict exists in one learnt conjunction, the same conflict cannot happen in another one. Hence, all conjunctions learnt are useful in terms of conflict resolution.

To improve the efficiency, we push the similarity computations into the tree building process. Therefore, we do not need to materialize the *scores* table. Furthermore, we do not need to construct all *scores* table columns if we have already obtained reasonable split point. The worst case complexity

Dataset	Cases	Classes	#cont attrs	#disc attrs
adult5K	5,000	2	6	8
adult	32,561	2	6	8
auto	205	6	15	10
cleve	303	2	6	7
crx	690	2	6	9
german	1,000	2	7	13
glass	214	7	9	0
hepati	155	2	6	13
horse	368	2	15	22
hypo	3,163	2	7	18
sick	2,800	2	7	22
waveform	5,000	3	21	0

Table 5: Dataset Descriptions

of this algorithm is $O(\mid R \mid \times d \times n \times (log(n))^2)$, in which $\mid R \mid$ is number of relevant cases, d is original feedback table dimension, and n is number of feedback. In practice, distinct relevant attribute values in an attribute (e.g., Loc) is much less than number of relevant feedback (i.e., $\mid R \mid$). Therefore, the complexity is close to standard decision tree complexity $O(d \times n \times (log(n))^2)$

4. EXPERIMENTS

In this section, we present the results of our experiments to evaluate the effectiveness of our query refinement method. We first present our experimental setup including user behavior model, the datasets we used, and a query generator that embodies the user model and generates queries appropriately. We then evaluate our new approach against four well-known algorithms extensively on the datasets. The goal of the evaluation is to establish our new algorithm as an accurate, and efficient relevance feedback refinement algorithm.

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 stored procedures in IBM DB2. We use IBM OSL package [10] as our linear problem solver.

4.1.1 Datasets and Similarity Functions

We used 12 datasets from the UCI machine learning repository [15], all of which have class labels. Table 5 shows the characteristics of the datasets.

There are two types of attributes in the datasets (i.e, continuous and discrete).

These datasets represent different domains of interests. For example, *german* dataset is in the financial domain, and *waveform* is more of the scientific knowledge discovery purposes.

We have consulted the documentations of each dataset and also the expert knowledge from the domains. For a discrete attribute, we analyze each pair of values for an attribute and decide the similarity value between them. For a continuous attribute, we use the formula in Section 2.1 to compute the similarity value for any two ranges of value.

4.1.2 Query Generator

To explore our query refinement method, we formulate two related queries, a *target* query q_t which represents a user's real information need, and an *initial* query q_i which represents his initial knowledge when starting the search. The query q_t can be more general or specific than q_i in terms of their 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 learning a specific query from a general query (i.e, G2S in our experiements). Conversely, if $q_t = q_2$ and $q_i = q_1$, we evaluate our algorithms on learning a general query from a specific query (i.e, S2G in our experiements).

The real life datasets used in the experiments are normally used for the classification tasks. The classification rules generated are often used to verify the existing domain knowledge. In our experiments, we use the classification rules to generate queries.

We chose an association rule [1] based classifier *Classification Based on Association* (CBA 2.1) [12] as a classification rule generator. This classifier has excellent classification accuracy and also generates all classification rules that satisfy minimum support and confidence thresholds. Our query generator generates 2 disjuncts for q_1 where each disjunct corresponds to a different classification rule generated from CBA. We choose two rules with high confidence (we use 85% confidence threshold to select rules). Normally, a high confidence rule corresponds to good domain knowledge. We use such rules to form the basis of the similarity queries, which normally are given by users. A rule generated by the classifier is a crisp rule. We convert the rule to a similarity conjunction by adding predefined similarity functions, and assign equal weights to all the condition involved. Once we have q_1, we derive q_2 by using the following generating rules:

(1) Randomly remove a predicate from each conjunction in the target query.
(2) Randomly disturb the weights in each conjunction by $\pm 10\%$ of each predicate weight.
(3) Randomly replace the query point in each predicate by one of its neighbor points. For ordinal attribute, we randomly choose a point having similarity value ≥ 0.9. For the non-ordinal attribute, we just choose the most similar value.

The first rule ensures q_2 is more general than q_1 in its DNF structure. The second rule makes the weights different. The third rule makes q_1 and q_2 related to each other, but not the same. We generate 20 query pairs (i.e., target and initial query) for each dataset for a total of 240 query pairs.

4.1.3 Algorithms Tested

We compare our new strategy to four existing ones. The first method is proposed in [20]. That focuses on learning conjunctions. We refer it as OCM (from authors' last names) in our experiments. OCM is similar to our approach. It also refines the initial query structure from a user's feedback. We also compare our approach to FALCON [28], which does not refine initial query explicitly, but uses the positive feedback to rank the data. We take Rocchio's method [24] as a standard vector based IR approach. In our experiment, we use attribute-value pairs as the vectors. Therefore, 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, we compare our approach to a well-known decision tree classifier C4.5 [23]. C4.5 takes the feedback data as a training set, and generates a classifier. It ignores the initial query structure, and predefined similarity measure on the data attributes. System uses this classifier

G2S Avg. Refine Time(sec.)			S2G Avg. Refine Time(sec.)		
New	*OCM*	*C4.5*	*New*	*OCM*	*C4.5*
0.4	0.8	0.2	0.5	0.8	0.3

Table 6: Average Refinement Time per Query (sec.)

to assign confidence levels as scores to the remaining data tuples.

4.1.4 Evaluation Process

We evaluate the above algorithms along the following metrics:
(1) Efficiency: the time taken to refine a query
(2) Effectiveness: precision and recall measures
(3) Simplicity of a refined query: number of the predicates used in the query

To evaluate effectiveness of a refinement algorithm, we simulate the desired user concept by executing a target query, and place top 20 tuples that satisfy the target query into a set R which we deem to be the *relevant* records.

Then, we execute the initial/original query (i.e., iteration 1). We compute the precision level at every 5% of recall interval until all the relevant tuples are retrieved. Note that we have 20 relevant points, every 5% recall interval means every relevant point retrieved. To study effectiveness of a refinement algorithm, we form the first learning set L by adding the top retrieved tuples containing two relevant tuples from the initial query result. The refinement algorithm uses set L as the *feedback* table to learn a refined query. Similarly, we update the learning set L by adding top retrieved tuples containing two new relevant tuples from the second iteration ranked list, and activate the refinement algorithm. In our experiments, we execute the refinement algorithm four times (i.e., five iterations) to evaluate its performance in different iterations. We believe it is good enough to evaluate our approach.

4.2 Results

For our technique to be useful in an online setting, it must be very efficient. As shown in Table 6, average execution time for a refinement cycle in our algorithm is around 0.5 seconds over a range of experiments with different datasets and queries. Due to space constraints, we cannot show the average refinement time in each dataset. The time is between 0.2 to 0.7 seconds. Note that Falcon and Rocchio algorithm do not generate queries explicitly, and are not included in this table. We notice that the new method runs more efficiently than OCM. This is because that OCM always starts structure tests, and it is more expensive than weight tuning. In our new approach, the activation procedure does not choose structure modification if there are no conflicts. Our new method runs slower than C4.5. This is expected since the feature space (i.e., scores table) used in our approach is larger than the original data feature space. In addition, our system also does weight tuning.

In Figure 3, we show the refinement effectiveness of the new approach and OCM on 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. Note that we only have 20 relevant points, and 5% recall interval means that for every new relevant point what the corresponding precision is. If a refinement algorithm performs well at a refinement iteration, it should have good precision level in all recall intervals. From the plots, we can clearly see our new approach performs much better than OCM in different iterations.

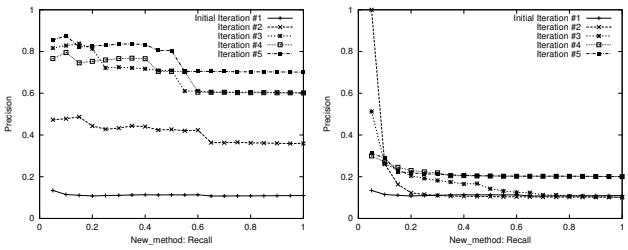

Figure 3: Precision-Recall: New method Vs. OCM

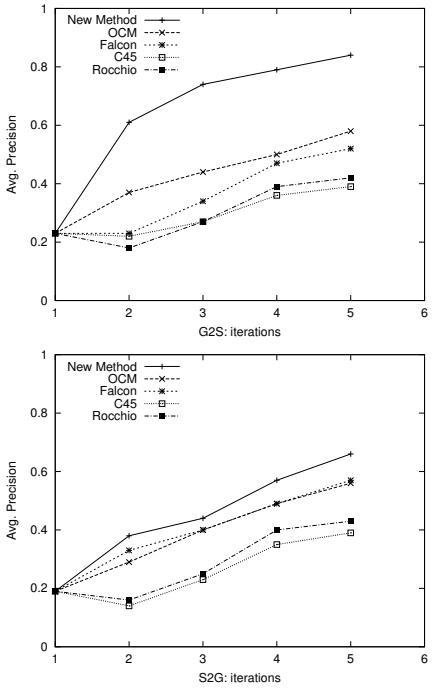

Figure 4: Precision Average at Each Iteration

Due to space constraints, we cannot plot all precision recall graphs for all the datasets and methods that we tested. We use the average precision at each 5% recall interval as a measure of retrieval quality. If a method does well in the iteration, this average should be high. For example, average precision for the adult data (G2S learning) at iteration 1 is 11%, and after one refinement cycle, our new method is able to achieve 40%. OCM still remains at 11%. Hence, our new method outperforms OCM after the first refinement cycle. By using the average precision measure, different algorithms can be compared directly in terms of the precision and recall.

Table 7 shows the detailed results of the general to specific learning (i.e., G2S) at the initial iteration (i.e., iteration 1), iteration 2 (i.e., after first refinement cycle), and iteration 5 (i.e, after last refinement cycle). From the experiments, we can clearly see that our new method outperforms the existing methods in almost all the datasets. Note that dataset *adult5K* is a sample dataset of *adult*. Our algorithm performs well even when data size increases nearly 6 times (i.e, from 5K to 32K). Due to space constraints, we cannot show detailed results for iteration 3, 4, and S2G part of results. To further evaluate the learning behaviors of each algorithm in

Dataset	It. 1	Iteration 2					Iteration 5				
	Init	*New*	*OCM*	*Falcon*	*C45*	*Rocchio*	*New*	*OCM*	*Falcon*	*C45*	*Rocchio*
adult5K	4%	25%	6%	1%	1%	2%	82%	15%	1%	10%	1%
adult	11%	40%	11%	10%	10%	1%	72%	20%	10%	10%	1%
auto	55%	79%	67%	69%	46%	47%	100%	100%	100%	100%	100%
cleve	41%	82%	73%	51%	44%	51%	93%	92%	100%	63%	81%
crx	8%	51%	33%	20%	3%	13%	84%	74%	44%	13%	15%
german	6%	74%	15%	10%	2%	13%	71%	51%	52%	2%	21%
glass	47%	79%	85%	29%	64%	11%	100%	100%	100%	100%	100%
hepati	36%	84%	74%	33%	47%	53%	100%	100%	100%	100%	100%
horse	21%	68%	26%	22%	14%	10%	100%	45%	64%	24%	34%
hypo	18%	72%	23%	11%	11%	6%	81%	45%	21%	21%	21%
sick	22%	62%	24%	13%	21%	13%	72%	24%	32%	21%	31%
waveform	5%	11%	10%	2%	1%	1%	54%	31%	2%	1%	1%
Avg.	23%	61%	37%	23%	22%	18%	84%	58%	52%	39%	42%

Table 7: Average Precision for G2S Learning at Iteration 1,2 and 5

different iterations, Figure 4 plots the overall average in the different iterations. We see our new method 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 initial query concept, it performs worse than our new method. The main reason is that the new method focuses directly on resolving the essential feedback conflicts, but OCM simply tries to fit concepts (i.e., predicates) that match to the feedback. For some cases, simple weight modification gives the correct ranking to the relevant tuples, but OCM may wrongly add in a set of predicates. Our new method outperforms OCM on average 24% in terms of average precision after first refinement cycle for the G2S learning. The average is computed over 240 query pairs, and this shows significant improvement.

Our new method and OCM outperform the other three methods in both *G2S* and *S2G* learning scenarios. It clearly shows the advantage 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.

Besides the precision and recall measure, another interesting measure of a refinement algorithm is the simplicity/complexity of its refined queries. Ideally, a refined query from a good refinement algorithm should be simple and effective in terms of retrieving relevant information. Table 8 shows the average number of predicates over the 240 queries at different iterations. Due to the space constraints, we only show the results after the first and the last refinement cycles. 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 our new method at different iterations are in between of initial and target average. However, OCM has about twice as many predicates as the target average. The main reason of this is because that OCM cannot represent disjunctive concepts explicitly, it uses additional predicates to simulate the disjuncts. In our algorithm, we directly deal with disjuncts in constructing *CSSs*. This gives us very close number of predicates as the target query. Since the new algorithm has good precision measure, this means the predicates are very close to the target predicates too. This increases the understandability since the refined query is not far away from what a user wants. 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, since there only very small number of positive feedback at each iteration, it generates very small number of rules with only one or two conditions. It seems very simple to understand, but since it has bad accuracy, the refined rules are not trustworthy. Falcon is not included because it does not generate query directly.

Dataset	G2S				S2G	
	Init	*It2*	*It5*	*Target*	*It2*	*It5*
New Method	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

Table 8: Avg. Number of Predicates in Query

5. RELATED WORK

The work most related to this paper is [20], in which authors proposed a query refinement framework on top of object-relational databases for learning SQL queries from user interactions. The refinement algorithms proposed were, however, ad-hoc and limited in the type of conditions that could be learned - they could only learn conjunctive queries.

There are two major limitations of the approach proposed in [20]. Firstly, it did not consider the 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 would add/drop wrong predicates from the query. In contrast, the approach proposed in this paper, attempts to estimate the benefit of structure and weight modification. An activation procedure invokes the appropriate learning algorithms when structure/weight refinement is expected to improve the query results. Secondly, [20] considered only learning a limited set of conjunctive queries. It did not support any mechanism to learn 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*. Furthermore, the approach would get confused and add other extraneous conjuncts that do not have much importance to the users' information need. This results in a suboptimal query with poor retrieval performance, as well as, a long query clause that is difficult to interpret.

Query refinement via feedback has been explored exten-

sively in the context of text document retrieval in the information retrieval literature [24, 27, 4, 2]. Generally, a vector space model is assumed (i.e., Rocchios method [24]). Refinement tasks focus on how to weight the elements in the vector space. There is no explicit query formulation, and attribute similarity measures. We can convert our refinement task to this model, but it performs badly in our experiments.

IR models have also been generalized to multimedia domain. Query refinement techniques have also been exploited in the multimedia domains, e.g., MARS [25, 14], Mindreader [11], FALCON [28]. 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. We can also convert our refinement task to this model, but since it 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. Works on DNF learners are closely related to our work. DNF learners in the literature can be grouped into two as divide-and-conquer based (e.g. decision tree learners [22]) and separate-and-conquer based or covering algorithms. Among the large number of algorithms in the latter category, the AQ family of algorithms [16, 17, 3], CN2 [6], PFOIL [18], and PRISM [5] are popular. In these approaches, predicates have no similarity semantics (i.e., crisp conditions). It is unclear how to incorporate initial query formulation and the attribute level similarity function directly into these approaches. Furthermore, these algorithms normally require large amount of inputs (i.e., training and testing ratio) before deriving good hypothesis. In our experiments, we directly use C4.5 in our refinement task, but it performs badly.

6. CONCLUSIONS

In many real life retrieval cases, a user normally has imprecise knowledge of what he wants. We provide a system to help him express the knowledge as a similarity search query, and refine this query continuously from the relevance feedback. In our new framework, we identified two tasks in similarity query refinement, namely refining query structure and determining the relative importance of predicates and disjuncts. We proposed a new refinement framework which controls the weight and structural learning. We implemented these algorithms and the extensive experiments we conducted show that our algorithm consistently outperforms previously suggested techniques in terms of retrieval quality and query simplicity.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of VLDB*, 1994.

[2] Ricardo Baeza-Yates and Ribeiro-Neto. *Modern Information Retrieval*. ACM Press Series/Addison Wesley, New York, May 1999.

[3] E. Bloedorn, R. S. Michalski, and J. Wnek. Multistrategy constructive induction: AQ17-MCI. In *Proceedings of the 2nd International Workshop on Multistrategy Learning*, pages 188–203, 1993.

[4] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems Applications*, Valencia, Spain, 1992.

[5] J. Cendrowska. PRISM: An Algorithm for Inducing Modular Rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[6] P. Clark and T. Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3(4):261–283, 1989.

[7] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *Proc. of the 15th ACM Symp. on PODS*, 1996.

[8] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[9] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*, May 2001.

[10] IBM. Ibm linear optimization pakcage: http://www-3.ibm.com/software/data/bi/osl/pubs/lpsol/lpuser.htm.

[11] Yoshiharu Ishikawa, Ravishankar Subramanya, and Christos Faloutsos. Mindreader: Querying databases through multiple examples. In *Proc. 24th Int. Conf. on Very Large Data Bases VLDB '98*, pages 218–227, 1998.

[12] B. Liu, W. Hsu, and Y. Ma. Integrating Classification and Association Rule Mining. In *Proceeding of KDD-98*, 1998.

[13] O.L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis. In *SIAM*. 1990.

[14] Sharad Mehrotra, Yong Rui, Michael Ortega-Binderberger, and Thomas S. Huang. Supporting Content-based Queries over Images in MARS. *Proc. of IEEE-ICMCS97*, 1997.

[15] C. J. Merz and P. Murphy. UCI Repository of Machine Learning Databases. http://www.cs.uci.edu/ mlearn/MLRepository.html, 1996.

[16] R. S. Michalski. On the Quasi-minimal Solution of the Covering Problem. In *Proceedings of the 5th International Symposium on Information Processing (FCIP-69)*. Vol. A3 (Switching Circuits).

[17] R. S. Michalski, I. Mozetic, J. Hong, and N. Lavrac. The Multi-purpose incremental learning system AQ15 and its Testing Applications to Three Medical Domains. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 1041–1045, 1986. San Mateo, CA: Morgan Kaufmann.

[18] Raymond J. Mooney. Encouraging Experimental Results on learning CNF. *Machine Learning*, 19(1):79–92, 1995.

[19] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Kriengkrai Porkaew, Sharad Mehrotra, and Thomas S. Huang. Supporting Ranked Boolean Similarity Queries in MARS. *IEEE Trans. Knowledge and Data Engineering*, 10(6):905–925, December 1998.

[20] Michael Ortega-Binderberger, Kaushik Chakrabarti, and Sharad Mehrotra. An Approach to Integrating Query Refinement in SQL. In *Proc. EDBT*, March 2002.

[21] Kriengkrai Porkaew, Sharad Mehrotra, Michael Ortega, and Kaushik Chakrabarti. Similarity search using multiple examples in mars. In *Proc. Visual'99*,

June 1999.

[22] J. R. Quinlan. Induction of Decision Tree. *Machine Learning*, 1:81–106, 2002.

[23] R. Quinlan. *C4.5: Program for Machine Learning.* Morgan Kaufmann, 1992.

[24] 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.

[25] Yong Rui, Thomas S. Huang, and Sharad Mehrotra. Content-based Image Retrieval with Relevance Feedback in MARS. *IEEE Proc. of Int. Conf. on Image Processing (ICIP)*, 1997.

[26] Yong Rui, Thomas S. Huang, Michael Ortega, and Sharad Mehrotra. Relevance feedback: A power tool for interactive content-based image retrieval. *IEEE Trans. Circuits and Systems for Video Technology*, 8(5):644–655, September 1998.

[27] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Computer Science Series, 1983.

[28] L. Wu, C. Faloutsos, K. Sycara, and T. Payne. FALCON: Feedback adaptive loop for content-based retrieval. *Proceedings of VLDB Conference*, 2000.