Cooperative Query Rewriting for Decision Making Support and Recommender Systems

Nader Mirzadeh
Bell ID
Stationsplein 45
3013 AK Rotterdam
The Netherlands
n.mirzadeh@bellid.com

Francesco Ricci
Free University of Bozen-Bolzano
Piazza Domenicani 3
39100 Bolzano
Italy
fricci@unibz.it

July 4, 2007
Abstract

This paper presents a new technology, called IQM (Interactive Query Management), designed for supporting flexible query management in decision support systems and recommender systems. IQM aims at guiding a user to refine a query to a structured repository of items when it fails to return a manageable set of products. Two failure conditions are considered here, when a query returns either too many products or no product at all. In the former case IQM uses feature selection methods to suggest some features that, if used to further constrain the current query, would really reduce the result set size. In the latter case, the culprits of the failure are determined by a relaxation algorithm, and explained to the user, enumerating the constraints that if relaxed would solve the “no results” problem. As a consequence, the user can understand the causes of the failure and decide what is the best query relaxation. After having presented IQM we illustrate its empirical evaluation. We have conducted two types of experiments, with real users and off-line simulations. Both validation procedures show that IQM can repair a large percentage of user queries and keep alive the human computer interaction until the user information goals are satisfied.

1 Introduction

Business to consumer web sites have proliferated, and nowadays almost every kind of product or service can be bought on-line. A huge market is available to the end-user through the World Wide Web. Nevertheless, a major problem is generated by this over-abundance of products and services, i.e., how to find the ”right” product that satisfies the user needs and wants, which is like “finding a needle in a haystack.”

This problem has only partially addressed by search engines (e.g., Google.com, Yahoo.com or MSN.com etc). A search engine can support only the initial stages of the search process, i.e., it can locate web sites where relevant products are sold. But search engines are keyword based and are not much useful within a web site to help the user to identify his preferred product in a catalogue. For this specific purpose, Recommender Systems, a new kind of information search and filtering technology, have been proposed [Resnick et al., 1994, Shardanaud and Maes, 1995, Resnick and Varian, 1997, Schafer et al., 2001, Adomavicius and Tuzhilin, 2005, Anand and Mobasher, 2005]. Recommender systems support personalized search, i.e., they help users to select products when large catalogues are available and the user has not enough knowledge to pick up autonomously the best option. For instance myproductadvisor.com asks a number of questions related to the intended usage of a PC, the user’s basic preferences about the PC price, brand and characteristics, and finally suggests a limited number of options. Or the well known book recommender system from Amazon.com suggests to a user books that were bought by users with similar buying behaviors [Linden et al., 2003]. In the next two sub Sections we shall first briefly discuss recommendation technologies to motivate the introduction of our proposed Interactive Query Management solution, that is discussed in the second sub section.

1.1 Recommendation Techniques and Failing Queries

Recommender systems implicitly assume that a user’s needs and constraints can be mapped, by means of appropriate recommendation algorithms, into product selections
using “knowledge” managed by the system. Many recommendation techniques have been proposed to automate this mapping [Adomavicius and Tuzhilin, 2005, Anand and Mobasher, 2005, Burke, 2002], and there are many deployed recommender systems for CD, movies, music, travel recommendation [MovieLens, 2007, Linden et al., 2003, Ricci et al., 2006a]. The most popular approach is called collaborative filtering which recommends products to a user based on his commonalities (e.g. product rates) with other users [Resnick et al., 1994, Shardanand and Maes, 1995, Breese et al., 1998, Sarwar et al., 2001]. Another commonly used technique is content-based filtering where the recommendation is based around the analysis of products previously evaluated by a user and the generation of a profile for a user based on the content descriptions of these products. The profile is then used to predict ratings for previously unseen items, and those deemed as being potentially interesting are presented to the user [Billsus and Pazzani, 1999, Balabanovic and Shoham, 1997, Pazzani, 1999].

While in the former approach (collaborative filtering) a user cannot specify any preferences and is basically pushed by the system with recommendations, in the latter the user is more in control of the recommendation process since in addition to the long-term user profile built by analyzing previously evaluated items, many content-based filtering systems allow the user to specify session-specific preferences to further constrain the result set. For instance, in case-based recommender systems, which can be considered as a particular type of content-based recommender systems, the user can initiate the recommendation process providing a set of preferences in the form of a partially defined product description [Burke, 2002, Bridge et al., 2006]. In content-based recommenders the user can actively seek for recommendation by querying the underlying catalogue, and the system role is to find products that match the explicit preferences specified by the user and other implicit preferences that may be guessed or derived using a history of previous users-system interactions. Nevertheless, in content-based recommender systems the search process could often meet two ”failure” situations: the system has not enough information to produce a small set of options or the user request is not satisfiable because some conflicting constraints have set or the item repository does not contain exactly the product sought by the user [McSherry, 2005].

Among the most interesting approaches proposed to tackle these problems we mention: logical filtering relaxation, similarity-based filtering and conversational recommender systems.

Logical-filtering imposes the exact match between the query and a set of products. In other words, the query expresses requirements as a set of constraints that must be satisfied. So, there may be situations where no product satisfies the query, and hence an empty result set will be returned. The latter is a major drawback of this technique, and there have been researches addressing this problem in the area of cooperative databases [Chu et al., 1996, Gaasterland et al., 1992, Godfrey, 1997]. Godfrey studied extensively the problem of empty result set for Boolean queries, i.e, where constraints are on Boolean variables [Godfrey, 1997]. He has shown that finding, one sub-query that returns some results requires a linear time, which is proportional to the number of constraints in the query, but finding all such sub-queries make the problem NP hard. Hence, query relaxation in logical filtering can be exploited in real applications only making some simplification of the general problem. For instance, one may search for an approximated optimal solution, where an optimal solution is that relaxing the minimum number of
Similarity-based retrieval is probably the most frequently used approach, in information retrieval [van Rijsbergen, 1979], database [Bruno et al., 2002], and case-based reasoning [Aamodt and Plaza, 1994, de Mantaras et al., 2006], to cope with the empty result set problem described above. Here a user query partially describes an ideal item/product, and the system retrieves from the catalogue those products that are most similar to the user’s ideal product described by the query. The major drawback of this approach is that when no product matches exactly the query, then the ranking induced by the similarity metric "compensates" automatically a match on some attributes with a mismatch on some others. For instance, assume that the user is searching for a hotel with car-parking facility in city center and with a certain cost. If there is no hotel satisfying all these three constraints, then the similarity retrieval could rank higher those hotels that match the parking and city center requirements, but with a higher cost, and rank lower those with the desired cost and parking but not located in the city center. But the system could even rank the products in the opposite way. This depends on the relative importance of the features and the particular local similarity measure used for each feature. The user is therefore not helped in understanding what are the available alternatives in order to take a decision autonomously. In addition, similarity-based retrieval cannot cope with the opposite situation when a user query matches exactly a large part of the catalogue. The induced similarity-based ranking will rank equally all these exactly matching products.

Conversational recommender systems are characterized by the fact that when faced with these failure problems they initiate an interaction with the user in order to refine the current query [Aha and Breslow, 1997, Göker and Thomson, 2000, Kohlmaier et al., 2001, Gupta et al., 2002, Shimazu, 2001]. These systems have mostly focussed on the large results set problem, and when the user query returns too many results these systems pose questions to the user to obtain additional preferences to further specify the current query. The questions are determined using feature-selection methods [Doyle and Cunningham, 2000, Kohlmaier et al., 2001, Shimazu, 2001], that exploit information theory principles (e.g. information gain [Quinlan, 1986, Witten and Frank, 2000]) to identify the attributes with higher discriminatory power. In other words, they select a feature and generate a corresponding question. When the user replies with a certain value a new additional constraint of the type "feature-value = user reply" is introduced, and this is supposed to narrow down the cardinality of the result set.

In Section 5 we shall discuss some other approaches and the relationships between our methods and those that have already been proposed.

1.2 Supporting Recommender Systems with Interactive Query Management

Considering the benefits and the limitations of both content and collaborative based filtering technologies, we have designed a novel hybrid collaborative/content based recommendation methodology, described in [Ricci et al., 2003, Ricci et al., 2006a]. In this methodology a unique human-machine interaction session is modelled as a case in a case-based reasoning framework [Aamodt and Plaza, 1994, Bridge et al., 2006]. A case collects all the information provided by the user during a recommendation session such
as the user’s queries to the product catalogues, the selected products, and, in case the user is registered, some stable user related preferences and demographic data. A recommender system based on this methodology models cases and products as two different concepts, and stores them in different repositories (case base and catalogues) This is in contrast with standard case-based reasoning recommender systems that view the catalogue of products as the case base (see [Lorenzi and Ricci, 2005, Bridge et al., 2006] for a discussion on that).

All input features provided by the user during an interaction session fall into two (not mutually exclusive) categories: content and collaborative features. Content features are those features that can be constrained in the users’ queries, and they are used as descriptors of the products in the catalogues. For instance, in a hotel description the hotel category and the presence of a garage are content features. On the other hand, the nationality of the user or the travel purpose are used to describe the trip, and are not part of any product description found in the catalogue. We have called such features collaborative, since these features are used to measure user similarity within a given session.

When a user searches for a product, she queries the corresponding catalogue in terms of a set of constraints representing the user’s needs and preferences. If the query fails to return a manageable set of products, then the user will be assisted by the Interactive Query Manager module (IQM) to refine her query. IQM is a conversational system that manages both the “no result” and “too many results” problems. When finally the user accepts a query, which is suggested by IQM, returning an acceptable set of items, these are ranked using a collaborative-by-content approach [Pazzani, 1999], which exploits a double similarity computation [Ricci et al., 2003, Ricci et al., 2006a]. The architecture, algorithms and evaluation of this IQM module is the core contribution of this paper, and will be described in the next sections.

In our approach ranking is based on implicit preferences and is performed after the explicit and attainable preferences are elicited by the user interaction with IQM. Hence IQM plays a major role in the recommendation process as a conversational and content-based product filtering component.

In this paper we describe the Interactive Query Management component (in Section 2) and in particular the query relaxation (in Section 2.4) and tightening (in Section 2.3) algorithms. Moving from an initial empirical evaluation [Ricci et al., 2003], which showed the classical entropy-base feature selection technique suggests features not correlated with user interests, we have introduced two new feature selection methods for query tightening. The first assigns to each feature a score proportional to the probability of being used by the user while the second assigns to a feature a utility value computed by combining the feature information content with the probability of being used by the user. We have therefore compared these two methods with the entropy-based one in two types of simulated user-system interactions (Section 3). The first one follows a more traditional approach that assumes the user will always reply to a posed question about a feature when he has a preference on the values of the feature. Whereas in the second evaluation, we assigned to each feature an acceptance probability that is proportional to the feature popularity measured in real user interactions with the system.

While in the first evaluation the entropy-based approach outperforms the new methods in term of result size reduction, in the second evaluation the situation is reversed and
the entropy-based method is outperformed by both the popularity and utility methods. The rationale of this result is that these two new methods select questions/features that will more likely be used to further constrain the result set and therefore they keep the interaction alive.

Query relaxation (described in Section 2.4) is implemented as an approximate linear time algorithm that if it can find a successful relaxation then that query would be close to a maximal one with respect to the number of constraints kept in the relaxed-query. Thus, since a query represents a user’s preferences, the algorithm relaxes as few as possible the user’s preferences. We discuss the properties of the proposed relaxation algorithm, and we show that it is powerful enough to recover from real failure situations most of the time (86% of the input failing queries).

At the end of this paper we briefly illustrate two recommender systems (Section 4) where we have applied the Interactive Query Management, we discuss some related work (Section 5), and we summarize our research contributions (Section 6).

2 Interactive Query Management

All major eCommerce web sites allow users to search in their product catalogues. These web sites ask the user constraints or preferences related to the products, and retrieve those items that match, exactly or only to certain extent, the user requirements. As we mentioned in the Introduction, two basic failure situations may take place in replying to a user query: either too many or none items/products are found. These situations occur quite often, even if sometimes the user is not aware of that because automatic filtering or implicit relaxation approaches as those described earlier are applied. For instance, when too many results match the user’s preferences/constraints, a common “trick” is to present only a few of them, by automatically filtering out some of the results. This filtering technique is typically not personalized, and this will tamper the goodness of the recommendation.

In our view the available solutions to these problems are not completely satisfactory. The underlying tradeoff between maintaining the user interface simple and at the same time catching all the user needs is typically resolved in real applications in favor of the simplicity, i.e., automatically solving the problem without involving the user in a dialogue. This is correct but it leaves the space open to more sophisticated solutions that can better balance simplicity with added functionality.

In fact, solutions to these problems can benefit by observing how these interaction problems are managed in real shops via human (customer) to human (clerk) interactions. In this situations we typically observe even long interactions, where questions arise from both the customer and the clerk to make explicit both the search conditions and the offers. In fact, the search constraints are never clear at the beginning, and the initially formulated customer preferences and requirements are iteratively refined, with the help of the clerk. For instance, in a travel agency a hotel search may proceed as follows.

- **Customer:** I’d like to find a 3 or 4 star hotel with the price up to 100 Euros.
- **Clerk:** There are too many hotels with these characteristics. Would you like to have facilities like car parking or air conditioned?
• **Customer:** Yes, I need car parking and restaurant.

• **Clerk:** I’m sorry but we do not have any hotels with all that characteristics. There are 40 hotels in the desired category with parking, restaurant, but you have to consider a price up to 110 Euros. There is also one hotel with your desired category and price which has a restaurant but it has no parking

• **Customer:** Let me see this cheap option [he sees it]

• **Customer:** Let me now see the expensive ones

• **Clerk:** Would you like to have Internet connection in your room?

• **Customer:** Yes [and she gets 10 suggestions]

What emerges from the above dialogue is that the clerk not only replies to the customer (decision maker), but also suggests modifications to the user’s requirements (query) to keep the conversation alive and better converge to a positive outcome, i.e., a user selection. In order to support such kind of dialogues, we have developed a technology called Interactive Query Management (IQM) that cooperatively and interactively helps the user to autonomously rewrite a query issued in the context of a decision making problem supported by a recommender system [Ricci et al., 2006a]. The next section describes the logical architecture of IQM.

### 2.1 Architecture of IQM

Figure 1 shows the logical architecture of IQM. In IQM the user’s information needs are encoded as a query. The client application, for instance a Graphical User Interface or a recommender component that ranks products, interacts with IQM providing the input query. IQM then displays items that satisfy the query constraints, if any, or otherwise shows suggestions to refine the query and collects further input from the user. The *Data Processing Engine* provides physical data storage, query processing, and some utility functions needed by IQM. When a query is sent to IQM, it is translated (according to some mapping rules) into a format that is processable by the external Data Processing Engine. This engine processes the rewritten query, and sends the required data (e.g., how many items do satisfy the query) to IQM to analyze it. At this point, the query result set may contain either: i) no item, ii) a reasonable number of items, or iii) too many items. Only in the second case the results are sent to the client. In the latter case, IQM calls the *Tighten* module to suggest a set of candidate features that should be constrained in addition to those already included in the query. In the former case (i.e., the query has returned an empty result set), IQM invokes the *Relax* module to find all relaxed alternative queries that changing or removing a small set of constraints in the failing query make the query satisfiable, and reports these alternative options.

In the next Section we shall first describe the query language, then the feature selection methods exploited in the Tighten component, and then the relaxation algorithm that computes the alternative relaxation suggestions.
2.2 Query Language

The query language should be simple to be used by a generic user but powerful enough
to let her to express the most important information goals. To support dialogues as
that shown in Section 2 the customer should be able to express her preferences as a
conjunction of constraints (e.g., constraints on price, category, parking etc.) that the
ideal product should satisfy.

IQM supports a vector based representation of products, and queries are defined as
conjunctions of simple constraints on a single features. More formally, let $X = \prod_{i=1}^{n} X_i$
be a collection type. We shall refer to a feature of $X$ as $x_i$, and $X_i$ will be called the
domain of $x_i$, for all $i = 1, \ldots, n$. A collection or catalogue $C_X \subseteq X$ is a finite set of
objects of type $X$. A query $q$ (over the collection $C_X$), is obtained by the conjunction of
atomic constraints; $q = c_1 \land \cdots \land c_m$ (or simply $q = \{c_1, \ldots, c_m\}$), where $m \leq n$ and each
c_j constrains a feature, say $x_{ij}$. Moreover, every $c$ in $q$ has one of the following forms,
assuming that $c$ constrains feature $x_i$:

$$
c \equiv \begin{cases} 
  x_i = \text{true} & \text{if } x_i \text{ is Boolean} \\
  x_i = v & \text{if } x_i \text{ is nominal} \\
  x_i \in [l_i, u_i] & \text{if } x_i \text{ is numeric} \\
  x_i = * & \text{for all feature types}
\end{cases}
$$

(1)

where "*" is a wild-card symbol and the constraint '$x_i = *'$ is always evaluated as true.

Note that even if the domain of a feature may be infinite such as the real numbers,
the feature values occurring in a catalogue are always finite. Hence, there are always
the minimum and maximum values for a numeric feature in a catalogue. The range of a
numerical feature is defined as $\text{range}(x_i) = U_i - L_i$, where $L_i$ and $U_i$ are the minimum
and maximum values of $x_i$ in the catalogue. In a range constraint, as $c = 'x_i \in [l_i, u_i]'$,
the range of the constraint, $\text{range}(c)$, is $u_i - l_i$, and we have $[l_i, u_i] \subseteq [L_i, U_i]$.

A constraint is called Boolean, numerical or nominal if it constrains a Boolean, nu-
meric, or nominal feature respectively. Let $\text{count}$ be the classical function that returns
the number of items that satisfy a query $q$ in the catalogue $C_X$, and let $\alpha > 0$ be an
integer. As we mentioned earlier we shall identify methods to cope with situations when
a query fails, i.e., the result set is empty ($\text{count}(q) = 0$) or when the results set is too
large, i.e., $\text{count}(q) > \alpha$.

We note that the definition of a constraint on a Boolean feature may seem redundant
since a Boolean feature is a nominal one as well, but in such a way we make explicit the
fact that Boolean features are constrained only with true values. Let us now consider an
example query.

Example 1. The query $q = \{(3 \leq \text{category} \leq 4), (\text{parking} = \text{true}), (\text{price} \leq 100), (\text{restaurant} = \text{true})\}$
models the query given by the customer at the second interaction step of the example
dialogue presented in the previous section.

2.3 Query Tightening

When a query returns many items the user will either browse the full list or will consider
only a subset of items, typically the top ones. A more convenient approach would be to
help the user to express additional preference(s) she may have and therefore to focus her
attention on a smaller and more manageable set of options. Hence, in IQM when a query returns a large result set, i.e., above an application dependent threshold, the system tries to assist the user to refine the query and build a more specific one. In such situation, IQM suggests some features, three in our recommender systems based on IQM, and waits for the user’s action. The user can either select/use some of the proposed features to further specify the query, or ignore the suggestions and browse the full list. In the following sections we present some feature selection techniques that have been integrated in IQM.

2.3.1 Feature Entropy

The entropy (or information value) of a feature is a measure of the amount of information provided by the feature for identifying a product [Witten and Frank, 2000]. Hence, for instance, if we want to find a hotel that has "TV" and "Parking" (Boolean features), then we may want to know if the first or the second feature is more useful in such an identification task. Imagine that all the hotels have TV but only half of them have parking facility, it is clear that the "TV" feature is of no use, its entropy is 0, whereas the second provides one bit of information, separating the set of hotels into two equally populated groups. Feature entropy and a derived measure called Information Gain have been largely used in building decision trees, as in the ID3 algorithm to select the best feature to further branch a node in the expansion of a decision tree [Quinlan, 1986].

Hence, similarly to its usage in the node selection problem in the induction of decision trees, the entropy of a feature can provide a score useful to rank the features not already constrained by the user. Features with higher scores should be asked to the user because they are the most useful to identify the required product. In fact, many conversational systems [Kohimaier et al., 2001, Shimazu, 2001] have applied this idea. But, in order to apply techniques similar to those used in induction of decision trees, these systems identify a particular feature as the ‘class’ of the product or consider each product as belonging to a different class. In these systems, when a query has a large result size, the feature having maximal information gain, i.e., the feature that maximally reduces the entropy of the product set, is selected and the user is asked to provide a preferred value for that feature.

The entropy of a feature, say $x_i$, on a given set of items, for instance the result set of a query, is given by the following formula

$$H^{i}_{R_q} = - \sum_{v \in X_i} p^i_v \log(p^i_v)$$

(2)

where $R_q$ is the result set of $q$, $X_i$ is the domain of $x_i$, and $p^i_v$ is the estimated probability of observing the value $v \in X_i$ in $R_q$. Here we estimate $p^i_v$ as:

$$p^i_v = \frac{\# \text{ of items in } R_q \text{ with } x_i = v}{|R_q|}$$

Filtering a result set by further constraining a feature with high entropy is likely to reduce the cardinality of the result set. The major problem of this approach is that it does not take into account the decision maker interests or preferences over different features. In other words, a feature may have a large entropy in the result set (for instance the "Parking" in the example above) but could be of minor interest for the user (he is
travelling by train). A better solution would be to integrate the usefulness of a feature with the likelihood that it will be interesting for the user. This observation is at the base of the feature selection methods described in the next sections.

2.3.2 Feature Popularity

While the feature Entropy, presented in the previous section, scores each feature according to a measure of the distribution of values in the repository of items, the method discussed in this section is based on a statistical analysis of the input queries collected by the system. The user queries represent the users’ preferences and needs. If a feature has been often used, i.e., it is ”popular”, then this feature is more likely to fall into the interests of a generic user of the system. More precisely, the popularity score of a feature \( x_i \) is defined as:

\[
\hat{p}_i = \frac{\text{# of queries constraining } x_i}{\text{total number of queries}}
\]

(3)

The computation of popularity is trivially simple and does not require an on-line computational effort (at query time), as the entropy does. Its major drawback, when used as scoring mechanism for feature selection, is that it may suggest features that are not very useful in reducing the result size of the current query, since its suggestion is solely based on users’ past queries and does not take into account the feature values distribution. In the above example, the popularity method may score high the ”TV” feature even if, knowing that the user is likely to be interested in TV, it is not useful in finding a good hotel for him, since all the currently selected hotels do have TV.

We argue that an effective feature selection method must balance ”popularity” and Entropy, suggesting features that would probably be used by the decision maker, hence keeping the conversation active, and those that would likely produce a good reduction of the result size. An approach to combine these two goals is provided in the next Section.

2.3.3 Feature Utility

We present here a third feature selection method that combines the information value, entropy, of a feature with an estimation of the probability of its usage by the decision maker, that is provided by feature popularity. We call this new method utility-based.

Utility theory is a tool for modelling rational decision making [Russell and Norvig, 2003]. In this setting, the utility function maps world states \( S_1, \ldots, S_M \) to real numbers, \( U(S_j) \), expressing the utilities (preferences) of the states. A state is the outcome of an action available to a rational agent \( A_1, \ldots, A_N \). The expected utility of action \( A_i \) given the evidence \( E \) is:

\[
EU(A_i|E) = \sum_{j=1}^{M} U(S_j)p(S_j|A_i, E)
\]

(4)

where \( p(S_j|A_i, E) \) is the probability of reaching state \( S_j \) after action \( A_i \) with the background evidence \( E \).

In our model, action \( A_i \) is performed by the system and models the suggestion of feature \( x_i \) to the user to further constrain her current query. If an item space has \( n \)
features, then we have $S_1, \ldots, S_{2n}$ outcome states ($i = 1, \ldots, n$):

$$S_{2i-1} = \text{user accepts } A_i$$
$$S_{2i} = \text{user rejects } A_i$$

When a system suggestion is accepted, the user specifies a preferred value for that feature, and the current query is tightened. In this context, given action $A_i$ only states $S_{2i-1}$ and $S_{2i}$ can be reached, hence $p(S_j | A_i, E) = 0$ if $j \neq 2i - 1$ or $j \neq 2i$, and $p(S_{2i-1} | A_i, E) = 1 - p(S_{2i} | A_i, E)$. Moreover, in this case $E$ is the evidence about the features already constrained at a given interaction stage.

We define the utility of state $S_{2i-1}$ as the entropy of feature $x_i$ in the current result set. The rationale is that a feature with larger entropy is more likely to reduce the result set if it is constrained. Whereas the utility of state $S_{2i}$ is 0, since the system suggestion is discarded by the user. Thus,

$$U(S_{2i-1}) = H^i_{Rq}$$
$$U(S_{2i}) = 0$$

(5)

where $H^i_{Rq}$ is the entropy of feature $x_i$, as defined in Eq. (2)

In order to compute the expected utility of action $A_i$, we must compute the probability $p(S_{2i-1} | A_i, E)$. This could be estimated by observing real interactions and measuring how many times the user accepted a system suggestion. A simpler estimate consists in using the popularity of feature $x_i$. Hence, we make the simplifying assumption that $p(S_{2i-1} | A_i, E)$ is proportional to the a-priori probability that $x_i$ is used in a query, i.e., is proportional to the Popularity of the feature $x_i$, defined in Eq. (3):

$$p(S_{2i-1} | A_i, E) = \beta \hat{p}_i$$

(6)

where $\beta$ is a positive real constant. The rationale of this estimate is that if a feature is frequently used/constrained in the users’ queries, then it is more likely to be accepted by the user when it is suggested by the system.

It follows from the above discussion and equations (4)-(6) that the expected utility of suggesting feature $x_i$ becomes

$$EU(A_i | E) = H^i_{Rq} \beta \hat{p}_i$$

(7)

We finally observe that when sorting a set of features according to their expected utilities, the $\beta$ constant does not have any influence since $EU(A_i | E) \geq EU(A_j | E)$ iff $H^i_{Rq} \hat{p}_i \geq H^j_{Rq} \hat{p}_j$.

### 2.4 Query Relaxation

We have already underlined the fact that a query expresses the user (customer) preferences about her ideal product. When a query fails, that is, the result set is empty, then IQM calls the Relax module (see Section 2.1) to find a set of relaxed sub-queries of the original failing query that return some results. These sub-queries are obtained by relaxing the minimum number of constraints, or in other words, keeping the number of user preferences as large as possible.
The function \( \text{relax}(c, \gamma, \delta) \) shown in Figure 2 defines the relaxation operator on a single constraint \( c \). This function takes as input a constraint \( c \), and two constants \( \gamma \) and \( \delta \) that are used only when the constraint is numeric. When the constraint is not numeric the relaxation is trivial, it simply discards the constraint. When the constraint is on a numeric feature, then this is a range constraint \( c = (x \in [l, u]) \), and \( \gamma \) specifies the factor used to decrease the upper bound, and increase the lower bound of the range \([l, u]\) (line 7 and 8 in \( \text{relax}(c, \gamma, \delta) \)). If the resulting change in the constraint range is too small, for instance when the user has set a very small range in the query, then we use the \( \delta \) parameter to force a range expansion, on both sides, at least of size \( \delta \). Note that \( \gamma \) could be \( \infty \), in this case, after the relaxation the range constraint become \( x = [L, U] \), or equivalently \( x = \ast \).

Note that \( \gamma \) and \( \delta \) are feature dependent parameters and should be tuned with the user sensitivity to changes in that feature. For instance we can assume that on a hotel price the minimum change in price that would make some difference to the user is 2 Euros, i.e., \( \delta = 2 \). Hence if the constraint \( (x \in [50, 60]) \) is relaxed and \( \gamma = 0.1 \) then, instead of creating a new constraint as \( (x \in [49, 61]) \), with a 10% relaxation on both endpoints, we generate the constraint \( (x \in [48, 62]) \). This method works particularly in case the user inputs an equality constraint in the form of a range constraint; for instance \( x \in [48, 48] \).

Having defined the notion of relaxed constraint we can say that a query \( q' \) is called a relaxed sub-query of \( q \) if and only if for all \( c' \in q' \) either \( c' \in q \) or exist \( \gamma, \delta \) and \( c \in q \) such that \( c' = \text{relax}(c, \gamma, \delta) \).

**Example 2.** Let \( q \) be the same query as in example 1, then \( q' = \{[3 \leq \text{category} \leq 4], [\text{parking} = \text{true}], [\text{price} \leq 110], [\text{restaurant} = \text{true}]\} \) is a relaxed version of \( q \), where the constraint on \( \text{price} \) is relaxed by 10 percent \( (\gamma = 0.1) \).

In the query relaxation procedure, which is presented below, we search for non failing relaxed sub-queries that differ from the input query only in one constraint. This is because from one side we want to minimize the changes to the original user request and from another side we must tame the combinatorial explosion of considering all the possible relaxations.

For this reason, when relaxing a single constraint does not repair a failing query, we try to remove two constraints but only if these two constraints have at least an "approximate" dependency relationship. In this sense, the relaxation procedure makes use of the notion of feature abstraction hierarchy, i.e., a hierarchy among some features of an item space. For instance, let us consider again the Example 2. In this query there are constraints on both category and price features. In our perspective the category feature is more abstract than the price since a constraint on the category is probably less informative when the user has already specified a constraint on the price, but the opposite is not true, since we may still have quite a large range of prices in the hotels of a given category. In fact, looking at the data, one can discover that the partition induced by the price, on the space of all the hotels, is approximately a subpartition of the category. In other words, the knowledge of the price greatly reduces the uncertainty over the category, i.e., the conditional entropy of the category given the price is low [MacKay, 2003]. In database theory one typically considers the functional dependency between attributes,
for instance between a key and another attribute [O’Neil and O’Neil, 2001]. Here we are referring to an imperfect or approximated functional dependency.

A hierarchy relation can be used in the relaxation process in a natural way; when two features linked by a hierarchical relationship are both present in the query, then the relaxation could start from the constraint on the feature with the lowest abstraction level. If relaxing that constraint does not produce any result, then it could be removed, and the constraint on the feature at the next-higher abstraction level is tried. In this way an abstraction hierarchy provides a heuristics for the relaxation process, i.e., to try first to relax the features with lowest abstraction, because this will make the lightest changes to a user query, and if this is not enough to proceed to a higher level. Going back to the example above, if both category and price are constrained then try first to relax the price constraint and then, if this is not enough, to relax the constraint on category.

An abstraction hierarchy can be defined by either mining the data and extracting functional dependencies or relying on background knowledge provided by domain experts. Functional dependencies should be rare, and more often probabilistic/approximate functional relationships between the features would be found. In our applications of IQM we have followed this second approach, but the proposed IQM relaxation is independent from the method used to generate the abstraction hierarchies, and therefore other methods can be applied as well. Let us now precisely describe the relaxation algorithm and how feature abstraction hierarchies are used.

First of all we denote with $FAH = \{F^1, \ldots , F^k\}$ a set of feature abstraction hierarchies, where each $F^i = (x^i_1, \ldots , x^i_n)$ is an abstraction hierarchy, i.e., an ordered list of features in $X$ (comparable features). We mean that $x^i_j$ is more abstract than $x^i_l$ if and only if $j < l$, and there are no common features in two different $F^i$s, i.e., features in different $F^i$s are not comparable.

Figure 3 depicts the algorithm used in the Relax module to find non failing relaxed queries of a failing query. The constraints in the query $q = \{c_1, \ldots , c_m\}$ are first partitioned in the set $CL = \{cl_1, \ldots , cl_{m'}\}$, where each $cl_i$ is an ordered list of constraints of $q$, such that the features in $cl_i$ belong only to one abstraction hierarchy $F^j$ (line 1). If the constrained feature in $c$ does not belong to any abstraction hierarchy, then a list containing only the singleton $c$ is built.

Let us make an example of how a set of constraints in a query is partitioned according to a feature abstraction hierarchy. Let $q = \{c_1, c_2, c_3, c_4\}$ be the query as in Example 1, and let the feature abstraction hierarchy be $FAH = \{\text{\texttt{(category,price)}}\}$, then $CL = \{(c_1,c_3),(c_2),(c_4)\}$ is the resulting partition of $q$ according to this $FAH$. RelaxQuery outputs at most one relaxed query for each element in $CL$. The first relaxed query is obtained by relaxing the constraints in $(c_1,c_3)$, the second by relaxing $c_2$, and finally the third is obtained by relaxing $c_4$. To find the first relaxed sub-query, the algorithm first checks if the relaxation of $c_3$ (price) is successful to produce some results, otherwise it removes $c_3$ and relaxes $c_1$.

More precisely, first a list of relaxation rates is initialized at line 3. The role of this array is to specify a sequence of $\gamma$ values by which a range constraint will be relaxed. The first and second entries are 0.1, which means that a numeric constraint will be first relaxed with $\gamma = 0.1$, then, if this is not producing a successful sub-query, it will be further relaxed once more with $\gamma = 0.1$ and then finally the constraint will be removed ($\gamma = \infty$). Hence, a numerical constraint will be modified at most three times by the
The loop at line 5 tries to relax the set of related constraints in each $cl \in CL$ while keeping the rest of the constraints in $q$ unchanged. The related constraints in $cl$ are relaxed starting from the constrained feature with the lowest abstraction by the while loop at line 9. $c$ is the constraint to be relaxed. The relax flag is used to control the loop of successive relaxations: only one relaxation for constraints on symbolic features, and a maximum number of times equal to $\text{gammas.size}$ (3 in this case) for the numeric ones. The $n$ variable (line 14) contains the result size of the relaxed query.

The while loop at lines 15-25 tries to relax $c$ with different ranges for a numerical constraint, or it iterates only once for a non-numerical constraint. The range in a numerical constraint is modified depending on the value of the current relaxation rate, i.e., $\text{gammas}[k]$. This loop terminates when the relaxation is successful, i.e., $n > 0$ (line 19). In this case the new relaxed query and the result size $n$ are inserted in the suggestion $s$ (line 20). If $n$ is still 0 then a new relaxation is executed (in case the constraint $c$ is on a numeric feature).

At the end of this loop, if a suggestion has been found then this is added to the suggestion list $SL$. Otherwise a new constraint taken from the same abstraction hierarchy should be tried. At the end of this process a suggestion list is returned. This contains the set (possibly empty) of relaxed queries and their relative result set sizes.

In fact the relaxation algorithm we developed can also manage a situation that can occur in the execution of the RelaxQuery procedure, namely that a numeric constraint is relaxed too much, producing a too large result set. In this case, before inserting the suggestion in the suggestion list (line 27) the system tries to reduce the range to a size that is in the middle between that causing the initial failure (empty result set) and the current too large result set.

The running time of RelaxQuery is $O(|q|)$ where $|q|$ is the number of constraints in the query $q$, since each constraint is considered a constant number of times.

## 3 Empirical Evaluation

In this Section we present the evaluation of the IQM algorithms. We have analyzed the performance of the Tighten and the Relax modules to understand how helpful they can be in assisting the decision maker (user) to solve the failure conditions we are considering. For the Tighten module, we have compared the performance of the feature selection methods presented in Section 2.3 in simulated user-system interactions. For the relaxation algorithm, we have evaluated the RelaxQuery algorithm (Section 2.4) on real failing queries, collected in a previous on-line evaluation of the NutKing recommender system with real users [Ricci et al., 2003].

### 3.1 Feature Selection for Tightening

In this section we compare the performance of the four feature selection methods introduced in Section 2.3. Each method, first sorts according to a score function all features not yet included in the current query, and then it selects the top scoring features. The score functions are: the Entropy, computed according to the feature values found in the
current result set (Eq. (2)); the Popularity (Eq. (3)); the Utility (Eq. (7)); and Random, as a base line for the comparison, which assigns to the feature a random score.

The above methods are compared by using a generic evaluation procedure that simulates user-system interactions. Each session is composed by some interactions. In each interaction the system suggests a predefined number of features \( n_f \), and the simulated user accepts or not at most one of them to further constrain the initial query. The value provided by the (simulated) user for the selected feature is taken from a test item. Thus, a simulation basically models a user that is trying to select the test item.

The evaluation procedure is listed in Figure 4 in details. The goal of this procedure is to run a set of simulated interactions and record: the number of items in the result set after each simulated interaction \( \text{count}(q) \) in the algorithm and the popularity of the feature used in the interaction (not shown in the algorithm). This procedure takes as input: a set of items (test set), the number of constraints \( n_c \) in the initial test query that should be created on the base of the test item, the feature selection method \( \lambda \) (i.e., one among Entropy, Popularity, Random, and Utility), the number of features \( n_f \) to be suggested to the simulated user at each interaction, the acceptance probabilities of features \( p \) and a parameter \( \text{iterate} \) that, if true, forces the simulation to try \( n_f \) additional features if none of the previous \( n_f \) were accepted by the simulated user.

At line 2, for each test item \( s \), an initial query is generated, which randomly constrains one feature, among the four most popular features, to the value found in \( s \). Then, if \( n_c > 1 \), the rest of the required constraints are generated by choosing randomly \( n_c - 1 \) features among those not constrained yet, and constraining them according to their values found in \( s \). If the feature is numeric then a range constraint is generated, i.e., a constraint of the form \( x_i \in [l_i, u_i] \). In our simulation the range is generated taking as lower (upper) bound \( l_i \) \( (u_i) \) the value found in the test item \( s_i \) and decreasing (increasing) it by 10\%, i.e., \( l_i = 0.9s_i \) \( (u_i = 1.1s_i) \).

After this initial step, the loop starting at line 6 simulates the user-system interactions. If the result size of the query is above the given threshold (set to 50 in this algorithm), then the feature-selection method \( \lambda \) is called to sort all the remaining features, i.e., those not constrained yet in \( q \). Then the while loop at lines 9-25 simulates a set of interactions with the user where the system suggests \( n_f \) features per interaction and the user is supposed to accept the first that meets his interests.

In fact, at line 10 the first feature in \( sf \) is selected and at line 11 there is the acceptance test. A feature \( x_i \) is considered accepted by the simulated user with probability \( p_i \), and its value \( s_i \) in the test item \( s \) is neither false (for boolean features) nor null (not known), hence a meaningful constraint can be generated. If a feature is accepted, then we consider this an interaction, increasing the interaction counter and exiting from the inner while loop.

Conversely, if a feature is not accepted we test if this is the last in a group of \( n_f \) features (line 16). If this is the case we increase the interaction counter. Moreover, if the \( \text{iterate} \) parameter is false, then we stop the simulation for test item \( s \). This simulates a user who has not replied to any of the first \( n_f \) (three in our experiments) suggestions and has aborted the IQM process. This is the interaction supported by NutKing. Then we increment the feature counter \( j \) in order to check a new feature, at the next iteration of the while loop (line 9).

The results of this evaluation algorithm depend on the feature selection method and
on the probabilities $p_i$. If all of them are 1, then this means that the user is always using a feature suggestion. Conversely if these $p_i$ are smaller than one then there is a chance that the simulated user will not reply, and the process can terminate anticipatively. These two situations are studied in the following sections.

In the two evaluations described in the next sections we have considered an accommodations catalogue containing 3400 accommodations,\(^1\) modelled with 15 features, 2 numeric and 13 Boolean.

### 3.1.1 Evaluation 1: User Always Accepts Feature Suggestions

Feature-selection methods for recommender systems have usually been evaluated with respect to the interactions length observed in simulated interactions [Aha and Breslow, 1997, Kohlmaier et al., 2001, Shimazu, 2001]. A typical simulation proceeds by first selecting a target product (test item in the FeatureSelectionEval() algorithm), and then searching for those products that are most similar to the target with respect to a subset of all the product features. If the target product belongs to the first $k$ items retrieved (stop condition), then the procedure terminates. Otherwise, a new feature is selected, the target product provides the value for a new constraint on that feature, and a new retrieval is performed. The stop condition may vary, for instance a variant version stops the simulation when at least one product, among the 10 products most similar to the target, is found in the first $k$ items retrieved. All of these procedures iteratively add one feature after another to narrow down the result set until the retrieval set contains an item considered as a good solution to the recommendation problem. The number of iterations is computed and better feature selection methods are considered those that produce shorter interactions.

As we have already mentioned, differently from the above quoted approaches that use similarity-based retrieval, we use logical based filtering in which every item in the result set satisfies the query constraints, and we stop the simulation when the result set size is less than 50. Using logical based filtering assures that the result set always contains the test item since this satisfies all the queries built using the test item itself.

In this first experiment we ran the evaluation procedure (Figure 4) on 500 items (the set $S$) randomly selected from the accommodation catalogue, with $n_f = 3$, i.e., three features are supposed to be suggested to the user at each interaction and with probability one ($p = \{1, \ldots, 1\}$) the user will use the first suggestion if a value for that feature is present in the test item. Otherwise, the simulated user will try the second, and then the third. Moreover, the iterate parameter is set to true, i.e., if none of these three features have been accepted then three more features are tried until all the features are considered. Hence, we are following here a more traditional approach which assumes that a posed question is always replied by the user [Doyle and Cunningham, 2000, Shimazu, 2001, Bergmann and Cunningham, 2002], and the simulation iterates until some feature can be used to tighten the current query.

Figure 5 shows the average result size of queries by different methods at each interaction and the average popularity of the feature used at a particular interaction. These results show that the entropy method outperforms the others with respect to the reduction of the result size. This is somewhat an expected result, in line with previous

\(^1\)These are real accommodations promoted by the local tourist organization in Trentino region
evaluation. The major problem of the classical entropy method is that it selects features that are not "popular" among the users.

Conversely the popularity and utility methods have better performance in term of popularity, which is quite obvious, but they cannot reduce the result size as fast as the entropy-based method. The Popularity method performs even worse than the Random one from the fourth interaction. This happens because the features selected in the fourth to sixth interactions are correlated with those selected in the earlier interactions and the newly built constraints are satisfied by most of the items in the catalogue. For instance, the parking feature is the fourth in terms of popularity, and 91% of accommodation items have the parking facility. We finally observe that the rise in average popularity (Figure 5 (b)) at the eight and ninth interactions for the Utility method is due to the fact that there are features with low utility because the entropy factor \( U(S_{2i-1}) = H^i_{Rq} \) in Eq. (4) is low but the popularity of the feature is not.

The conclusion of this evaluation is that the classical Entropy method is the best option when the user is supposed to always accept the suggested feature and constrain accordingly the result set. But what would happen if we explicitly model the fact that the user does not always reply to all question posed by the system? The following section presents the evaluation of the proposed feature selection methods taking into account the likelihood of a user reply.

### 3.1.2 Evaluation 2: User does not Always Accept Feature Suggestions

In this section, we evaluate again the proposed feature selection methods but with a different simulation of user-system interaction. Here we assume that with a non zero probability the user does not reply to the system feature suggestions, and if the initial set of suggested features (three in our example) are not accepted, then the simulated interaction is stopped.

As discussed in Section 2.3.3, the probability \( p_i \) that a user accepts a suggested feature, say \( x_i \), can be estimated as \( p_i = \beta \hat{p}_i \), for all \( i = 1, \ldots, n \). If we want to realistically simulate user behavior in an off-line evaluation of the proposed feature selection methods then we must estimate the constant \( \beta \). The \( \beta \) value must produce \( p_i \) values that are close to the real acceptance rates that can be observed with human subjects.

It is worth mentioning that \( \hat{p}_i \), i.e., the popularity of feature \( x_i \), is the a-priori probability that a user constrains that feature in the initial query, and it is likely to be an over-estimate of the acceptance probability of using \( x_i \) in a successive interaction. This is because the more features the user has already constrained the more unlikely is that he further constrains other features.

In an experiment we conducted with real users (see Section 4.1 for more details), the Entropy method was used in the Tighten module, and we observed that the suggestions were accepted by the user only 28.6% of the times. This rather low acceptance rate can have many explanations. For instance, the graphical user interface might be too complicated and the user could not understand the suggestions, or the suggestions did not fall into the user’s interests and were ignored. Whatever be the right interpretation, this is irrelevant to our evaluation, provided that the simulated acceptance rate is close to that observed in real interactions.

In order to estimate \( \beta \) and obtain acceptance rates \( (p_i = \beta \hat{p}_i) \) close to the observed
ones, we ran simulations with different values of $\beta$, and we found that with $\beta = 0.5$ the overall acceptance rate of the Entropy feature selection method comes close to 28.6%. Then, we have used this acceptance model, i.e., $p_i = 0.5\hat{p}_i$, for all the methods, implicitly assuming that a feature is accepted or not with a probability that depends only on the feature itself and not on the other features suggested by a particular method (in previous interactions or in the same interaction).

We ran the evaluation procedure again with 500 randomly selected items, $n_c = 1$ (one constraint in the initial query), $n_f = 3$, i.e., only 3 features were suggested at each interaction, and \textit{iterate} set to false. That last condition means that if none of the three suggested features is accepted, then the interaction stops. Finally, as we have mentioned above, the acceptance model now is not trivial as before ($p_i = 1$), but $p = (0.5\hat{p}_1, \ldots, 0.5\hat{p}_n)$.

Figure 6 shows the results of the simulations under the new settings. The Random method now performs worse than all the other methods for all interactions, which is much more meaningful than before. The pure entropy method does not reduce the result size much after the second interaction. On the other hand, the Utility method, and more clearly the Popularity method, which performed poorly in the previous simulations, are now the best. This is because they select features that may not be so effective in reducing the result set, but have higher probability of being used, hence the simulated interaction proceeds. We must observe that in these graphs we assume that if a simulated interaction is stopped, for instance at an interaction step with $M$ items in the result set, then in the next interaction step the result set is still of size $M$.

In a successive evaluation we checked if the results shown before were due to a particular catalogue. Hence, we considered two additional catalogues used in another recommender system DieToRecs [Fesemmaier et al., 2003, Ricci et al., 2006b]. DieToRecs is described in Section 4.2, and, as we did for NutKing, we mined the real queries made by a population of users that tried the system prototype, and we computed the statistics required to measure the Popularity and Utility score mechanisms. The two additional catalogues are here called DTR-Destination and DTR-Lodging. The former is a catalogue of 600 locations in South Tirol (Austria) and Trentino (Italy). The latter is a catalogue of 20870 accommodations in the same regions. These catalogues contain real items promoted by TisCover.com. Also in these two trials we saw that the Entropy method performs worse than the Utility and Popularity. Here in these examples Popularity behaves worse than the Utility. Looking more deeply in the evaluation results we found that the reason was that these methods had almost the same acceptance rate in the simulations. But, since the utility of a feature is proportional to its entropy (Section 2.3.2), a high scored feature by the Utility method has relatively large entropy (in the result set) and it is likely to reduce the result size of the query better than a feature highly scored by Popularity. In other words, in these examples, the Utility method can better balance usefulness of the feature, in term of result set size reduction, and probability of being used.

A final observation is related to the computational cost of these methods. Both Entropy and Utility methods require to compute the entropy of the remaining features on the result set and this has a linear time cost in the number of items in the results set. Conversely, the Popularity method is a constant time (inexpensive) method.
3.2 Query Relaxation Evaluation

In this section we present the empirical evaluation of the RelaxQuery procedure introduced in Section 2.4. From a previous empirical evaluation of the NutKing recommender system with real users we derived a set of real queries that failed to retrieve any item because they were over constrained. We have therefore examined the performance of the relaxation algorithm on these queries, and we have measured in how many cases RelaxQuery was able to assist the decision maker explaining the cause of the failure. We shall also show an estimation of how difficult it would be for a user to find a successful (i.e., non-empty result set) relaxed-query by relaxing a minimum number of constraints without any help from the interactive query management (IQM).

The evaluation here discussed is based on four catalogues: accommodation, event, location, and sport activities. We collected all the users’ queries submitted to different catalogues during an empirical evaluation of the NutKing system with real users (Section 4.1). Let’s call $Q$ this query set, and let $FQ \subseteq Q$ be the subset of failing queries, i.e., those returning a void result set. Then we ran the RelaxQuery algorithm on each query $q \in FQ$ to compare the subset of failing queries, say $SQ$, that the RelaxQuery could find a successful relaxed-query, with and without using the notion of feature-abstraction-hierarchy (FAH) (Table 1). The relaxation algorithm that does not exploit FAH is described in [Ricci et al., 2002], and is the algorithm used in the NutKing evaluation mentioned above. This is basically similar to RelaxQuery here presented but does not group the query constraints according the FAH subsets, and hence the generated relaxations modify only a single constraint in the original failing query.

In total, (see Table 1) using FAH the proposed algorithm was able to find a successful relaxed-query 83% of the cases (232/279), whereas the same algorithm not using FAH was successful 61% (172/279). It is worth noting that this is obtained by loosing a property of the previous method, i.e., the capability to find all the successful relaxed-queries that relax only one constraint. In other words, there are cases in which using FAH two constraints in a hierarchy are relaxed even when only one is necessary. This happens when relaxing the constraint on the more abstract feature, and still keeping the constraint on the less abstract feature would give some results. However, this situation is very unlikely since an abstraction relationship between $x$ and $y$ is an approximate functional dependency. In fact, let’s assume that $y$ is functionally dependent from $x$, i.e., there is a function $f_{xy}$ from the domain of $x$ to the domain of $y$, such that $f_{xy}(v_x) = v_y$ for each tuple $v$ in the database, where $v[x] = v_x$ and $v[y] = v_y$ are the values of the attributes $x$ and $y$ in the tuple $v$. If we consider a query containing the constraints $y = v_y$ and $x = v_x$, with $v_y = f(v_x)$, then there cannot be a tuple $t$ such that the value of the more abstract feature does not satisfy the constraint, i.e., $t[y] \neq v_y$, and the value of the less abstract one satisfies the constraint, i.e., $t[x] = v_x$. Therefore, if the abstraction relationship is close to a functional dependency and the user is not generating queries in a random way, then our approach should keep the number of relaxations close to the minimum.

The Figures 8(a)-(d) compare the size of the sets $Q$, $FQ$, and $SQ$ for the location, accommodation, event, and sport catalogues, respectively, when FAH is used. Figures 8(a)-(d) show that RelaxQuery can always find a successful sub-query when the query (in our sample) contains up to three constraints. This number is even higher for
the queries submitted to the event and lodging catalogues. In particular, considering the accommodation catalogue, we observe that more than half of the queries failed to return any item, and the RelaxQuery could find a sub-query 83% of the times (Table 1).

Finally, we want to illustrate how difficult it would be for a user to refine autonomously her failing query. The discussion here is hypothetical, and it is aimed at computing, in the worst case scenario, the number of attempts required to repair a query by discarding a minimum number of constraints.

Let \( q = \{c_1, \ldots, c_m\} \) be a query, and assume that the RelaxQuery method can find \( k \) successful relaxed-queries. The maximum number of attempts the user has to do in order to find a successful change to her failing query by relaxing only one constraint is \( m - k + 1 \). In fact, the user after getting an empty result set, will remove one constraint, and she will submit the new query (second attempt). If this query fails again, then the user must set back the removed constraint and discard another one. This process will continue until the user receives, in the worst case, some results after \( m - k + 1 \) attempts. Actually, this number will be even larger if a query constrains a numerical feature, and the user wants to try different ranges for that feature. This is obviously unfeasible in reality. A special case is when \( k = 1 \), here \( m \) attempts are required in the worst case. This is illustrated in Fig. 9 with a dashed line. The straight line in the Figure shows the actual average worst case \( m - k + 1 \) computed on the sample of queries submitted to the accommodation catalogue during an empirical test (see Section 4.1). The dotted line shows that a constant number of attempts is required if the user follows IQM suggestions.

4 Recommender System Applications

The interactive query management technology described in this paper has been applied in two recommender systems: NutKing and DieToRecs. These systems provide recommendation for single products and for bundles of travel products (e.g., accommodation plus a destination and some event). These systems are driven by a recommendation methodology called Trip@dvice, that integrates case-based reasoning with interactive query management [Ricci et al., 2003] [Ricci et al., 2006a]. Case-based reasoning (CBR) is a problem solving methodology that tries to solve a problem at hand by using the solutions of past similar problems [Aamodt and Plaza, 1994, Watson, 1997, Bridge et al., 2006]. We have applied CBR for ranking (recommending) products that explicitly meet a user’s preferences (query), by deriving implicit preferences contained in previously stored user-system interactions (cases). To achieve this goal first IQM is used to support the retrieval of a manageable set of items that meet completely the user’s needs. Then, CBR is exploited for ranking higher those retrieved products that are more similar to those selected in similar search contexts. We illustrate in the following sections two prototypes built using this methodology focusing on the usage of IQM.

4.1 NutKing

NutKing recommends travel products and services (items) that the regional tourism organization of Trentino (Italy) promotes [Ricci and Missier, 2004]. When a query to a product catalogue (e.g. accommodations) returns too many items (i.e., above 15), NutKing suggests three features, and it asks the decision maker (user) to provide her
desired values for some of those features. For instance, Figure 10 shows, on the left, the form where a user inputs search conditions, here for a 3 or 4 star category hotel with the price ranging from 18 to 42 euros in Trento area. The system finds items satisfying the query, and in this case suggests to the user to specify one or more features. Here the suggestions are: 1) TV, 2) close to the city center, and 3) if the user has pets (right side of Figure 10). The user can specify any of those 3 features, or constrain other features, or browse the full list of 16 accommodations.

In the following snapshot (Figure 11), we see that the user has further specified: TV, close to the city center, and restaurant. Note that the restaurant feature was not suggested by the system, but the user has added it anyway. The system computes again the result size of the new query and finds that there is no hotel satisfying the user’s query. In this case, the system suggests to relax some constraints and explains that removing these preferences the user will get some results.

We conducted an empirical evaluation of NutKing (see [Ricci et al., 2003]), and we discovered some interesting results related to the interactive query management. We note that in NutKing the Entropy method was used by the Tighten module (Section 2.3.1) and the RelaxQuery method did not support the notion of feature abstraction hierarchy (FAH).

On average 4.4 features were constrained in a user query and 13.4 queries were issued by a user in a recommendation session. For 6.3 queries (47% of the total user queries) the system suggested some query relaxations, and in 2.8 cases (45% of the suggestions) the user accepted a suggested relaxation. We see this acceptance rate of the relaxation suggestions a good result, considering that the user behavior is often erratic and not always focused in solving the task.

On average the system suggested in 2.1 of the user queries (15.7%) to tighten the query, and only in 0.6 queries (28.6% of the suggestions) the user accepted to tighten the query using one system suggestion. We hypothesized that it was because of the Entropy based feature selection method, and this motivated the new feature selection methods proposed in this paper. For instance, in the example shown in Figure 10, the system suggests to specify if the user has pets, because (roughly) half of the accommodations in the result set allow guests to bring their pets and half do not, hence producing a large entropy score for that feature. But a few guests do have pets and are interested in replying to that question.

### 4.2 DieToRecs

DieToRecs (Intelligent Recommendation for Tourist Destination Decision Making) is a second recommender system that uses IQM [Fesnmaier et al., 2003, Ricci et al., 2006b]. DieToRecs extends NutKing and supports multiple decision styles, one based on Interactive Query Management and ranking based on case-based reasoning, and another called recommendation-by-proposing [Ricci et al., 2005], first introduced by Shimazu in his ExpertClerk system [Shimazu, 2001]. Here we focus on IQM usage in DieToRecs.

Figure 12 shows the IQM in action guiding the user to recover from a failing query that requires tightening. In that figure it is shown the system reply to a query for a 3 or 4 star category hotel and price in the range 18 to 42 Euros that is returning too many items. IQM suggests the additional features: 'boarding' (i.e., breakfast or half board) and 'near
city center’. Note that in DieToRecs the user can select only one feature. If the query still returns a large result set, then three additional features are suggested. This process continues until the user either receives a small ranked set of items or decides to look at all the items in the result set. If the user wants to add all the suggested features at once then she has to follow the link to another page to edit her accommodation preferences (query).

Figure 13 illustrates the situation where the user has added the features ‘near city center’, 'TV', and 'restaurant’ to the previous query. Now there is no hotel in the catalogue that satisfies the new query, and the system suggests to relax the TV feature if the user wants to received some results.

DieToRecs has been empirically evaluated with real users as well and compared with other systems. We refer the reader to [Zins et al., 2004a, Zins et al., 2004b] for more details on this evaluation.

5 Related work

Dealing with failing queries has been largely investigated in different studies either when a query returns a large result set, or when an empty result set is found. Not much work has been dedicated to dealing simultaneously to both problems. Hence, we illustrate here some related research on feature selection techniques dealing with large result set queries, and research devoted to address the empty result set problem, that extend the discussion presented in Section 1.

Many researches have been dedicated to attribute-selection techniques in case-based reasoning. Most of the techniques were developed within CBR applications for diagnosis and classification, and more in general in conversational case-based reasoning systems [Göker and Thomson, 2000, Gupta et al., 2002]. In these applications it is likely that the user would continue to answer the questions until she receives a solution of the problem. This seem not to apply in web-based eCommerce applications where the user is only one click away from a competitor web site and get frustrated very easily if prompted with apparently unnecessary or not interesting questions. Thus, in eCommerce applications, the likelihood of a user reply has to be taken into account, and this has not appropriately considered in the quoted works.

Doyle et al. [Doyle and Cunningham, 2000] proposed an attribute selection technique for eCommerce applications based on (conceptual) clustering of the product catalogue. The idea is to select the feature at each step that best identifies a distinct cluster on the result set of the query. Shimazu [Shimazu, 2001] has proposed an interaction model between a computerized salesclerk and a customer where the system can propose some products, in order to receive feedback from the user, or pose explicit questions selected using an entropy-based scoring mechanism, until a manageable set of products is reached.

Our proposed feature selection method and its evaluation follows the work initiated by Kohlmaier et al. [Kohlmaier et al., 2001]. However, there are some differences. The most important is that they use similarity-based retrieval, while in our approach a query is a logical filter containing a set of constraints that must be satisfied. Moreover, in our approach, the entropy of a feature is calculated on the result set of the query, while in their approach the computation is done on the whole catalogue. Both approaches
try to simulate the question acceptance rate of the user, but the probability that a user will accept a tightening suggestion is estimated, in our case, by mining the log data of real user interactions, while in their work the knowledge of a domain expert is exploited. Other changes pertain to the precise evaluation strategy that simulate user-system interactions. We stop a simulated interaction if there is no reply to the posed questions (i.e., none of recommended features is accepted/used by the simulated user to further specify the query), while in their simulation the process continues until all questions have been answered. This approach, as we explained in this paper, is clearly in favor of the entropy-based methods, but it is far from modelling the observed user behaviors.

The research on queries with empty result set goes back to the work of Kaplan [Kaplan, 1982], where he describes a system that supports a cooperative query answering to queries expressed in natural language. He argues that it is more informative to let the user to understand the cause of failure, and to provide answers that partially satisfy the query, than just reporting a void answer like the empty result set.

Motro introduced the Flex system, a natural language front-end to query processing in relational databases [Motro, 1990]. Among many other properties, Flex knows how much to relax a numeric constraint. For instance, the constraint in \( q_0 = \{\text{category} \leq 2\} \) must be relaxed by 1, so when the query fails it is relaxed to \( q_1 = \{\text{category} \leq 3\} \). In our methodology, numerical constraints are relaxed without using a-priori knowledge. Flex can compute all the minimal generalized failing queries. For example, if \( q_1 \) still returns no result, but \( q_2 = \{\text{category} \leq 4\} \) returns some results, then \( q_1 \) is a minimal failing query of \( q_0 \). These queries are the most relaxed ones that still fail, hence provide an indication of the “cause” of the failure. This concept has been used also by McSherry [McSherry, 2003, McSherry, 2005] that is discussed below. But unfortunately finding all such minimal failing queries is an NP-hard problem [Godfrey, 1997]. In contrast, we try to compute the maximal succeeding relaxed queries with the limitation of searching these queries only in a subset of all possible relaxations. The subset is given by the sub-queries differing only on one constraint or in a group of constraints that are “approximately” functional dependent (abstraction hierarchy).

Value abstraction was introduced in the CoBase system [Chu et al., 1996], where an abstraction hierarchy among feature values was built, and the relaxation operators could move up and down in the hierarchy. In our approach, a hierarchy is a relationship between features (e.g., country\(\rangle\)county\(\rangle\)city), derived from domain knowledge or from the analysis of the data, and is exploited to find successful relaxed queries of a failing one. While building a values abstraction hierarchy is quite complex and costly in CoBase, in our approach the feature hierarchy can be achieved with limited effort.

Gaasterland et al. describe a logic-based approach for developing a cooperative answering system that accepts natural language queries, and can explain the cause of any failing query [Gaasterland et al., 1992]. Godfrey has extensively investigated the problem of identifying the cause of a failing Boolean query [Godfrey, 1997]. He has derived an algorithm (called ISHMAEL) to find successful maximal sub-query (called XSS), i.e., those not contained in any other successful sub-query. He shows that finding only one XSS can be done in linear time proportional to the number of constraints in \( q \), but finding all XSS is intractable. The major difference between RelaxQuery and ISHMAEL relates to the search mode. While the latter does a depth-first-search in the query lattice to find
an XSS, the former extends the lattice of sub-queries since the numerical constraints are not removed but relaxed to some extent, and the search is restricted to find queries that relaxes a minimum number of constraints. Hence, the latter is incomplete to avoid the combinatorial explosion of the search space.

McSherry [McSherry, 2003] has approached the relaxation problem differently by looking at the products that do not satisfy a given query. For each product in the catalogue the set of attributes that do not satisfy the query (called compromise set) is computed. Products are grouped according to the compromises done, and a product/item (called case) representative of the compromise and having highest similarity to the query is put in a retrieval set. The problem with this approach is that the set of all possible compromises is exponential in the size of the query, and there is no indication about how to cope with such a problem.

Burke described a different approach to interactive query management called navigation by critiquing that implements a particular approach to automatic query relaxation [Burke, 2000, Burke, 2002]. The user starts the interaction with the system either mentioning a known product (restaurant) and asking for a similar one, or selecting a set of high-level features (case features) and searching for a product that matches those features. With this input information, the system first selects from the database the set of all products that satisfy the largest number of logical constraints generated by considering the input features type and value. The system, if necessary, implicitly relaxes the least important constraints until some restaurants could be retrieved. Then the system sorts the retrieved cases using a similarity metric. This similarity metric assumes that the user goals, corresponding to the input features, could be sorted to reflect the importance of such goals from the user point of view. If the recommended product satisfies the user then the interaction finishes. But if the user is not satisfied, because of some features of the proposed product, then he can criticize them. If, for instance, the price is too high and the user is looking for something cheaper, then she can “tweak” the original request and provide a new input explicitly mentioning that the result must have a cheaper price. This starts a new recommendation cycle, and the criticized features is considered the most important user goal.

In this way, explicitly critiquing a feature means that the user’s constraints on other product features will (implicitly) decrease their importance. In our approach, the user has complete control of the relaxation (and tightening) process with his preferred value(s), and at the same time the system keeps the maximum number of the user’s constraints in the relaxation process.

6 Conclusions

We have presented a front-end query processing methodology for recommender systems that copes with queries that return either a large result set or no results at all.

Moving from an initial empirical evaluation [Ricci et al., 2003] that showed that the entropy-based feature selection technique suggests features not necessarily interesting for a user, we have introduced two new feature selection methods. The first assigns to each feature a score proportional to the probability of being used by the user, and the second assigns a utility value to a feature combining the feature information content
and the probability of being used by the user. We have therefore compared these two methods with a previous one, the entropy-based, in two types of simulated evaluations. The first one follows a more traditional approach that assumes the user always replies to a posed question/feature when he knows the value of the feature. In this case, the entropy-based method outperforms both proposed ones, because it selects features that effectively discriminate the underlying data source.

In application domains such as diagnosis, the assumption that a user always replies to a posed question is reasonable. But that assumption is not acceptable in eCommerce applications and in recommender systems where the user may not be interested in the additional product features proposed by the system [Anand and Mobasher, 2005, Adomavicius and Tuzhilin, 2005, Bridge et al., 2006]. Therefore, in a second evaluation, we assigned to each feature an acceptance probability proportional to the feature popularity, measured in real user interactions with the system. Moreover, we tuned the overall acceptance rate in the simulations to be close to that measured in the user study. This second evaluation shown that, in case the probability of accepting a question is considered, the two new methods perform better in reducing the result size, because they select questions/features that will more likely be accepted, and therefore they keep the interaction alive. The proposed methods are very simple to implement, but they require some background knowledge of the user behavior, such as, features’ popularity, and features’ probabilistic dependency.

We have also presented in this paper an approximate linear time algorithm for query relaxation. If a solution is found, then this is close to be maximal with respect to the number of constraints kept in the successful relaxed query. Thus, since a query represents user’s preferences, the algorithm relaxes as few as possible the user’s preferences. This algorithm is simple enough to be easily integrated into any eCommerce application. An empirical evaluation of the algorithm is here presented, and it is shown that it is powerful enough to recover from real failing situations most of the times (86% of the input failing queries).

The current algorithms supporting the tightening and relaxation modules still have a number of limitations. As we have mentioned above the relaxation algorithm cannot find all the possible relaxations, and those found could be not the best. We plan to investigate other techniques that could find more relaxed successful sub-queries still not too far from the optimal ones (minimum constraints relaxation). This can only be achieved exploiting heuristics driving the search for subsets of constraints to remove/relax, towards those that are the more likely cause of the failure. It is also important to note that not all the relaxations are equally preferable, hence an extended algorithm should integrate information about the more relevant or important features for the user.

This observation makes the relaxation and the tightening problems more close together. In fact, in both cases we are investigating new ways to incorporate a user model into the relaxation/tightening algorithms. The more user preferences could be taken into account the more likely is that the proposed suggestions will be used, even if these are not optimal with respect to the minimization of the number of constraints relaxed or tightened. This makes this area an interesting and promising field for application of user modelling [Billsus and Pazzani, 1997, Fisher, 2001, Webb et al., 2001] and decision analysis [Clement, 1996, Russell and Norvig, 2003].
References


Table 1: Queries submitted to the catalogues. Q is the total number of queries, FQ is the subset of failing queries, SQ (SQ FAH) is the subset of FQ that can be successfully relaxed without (with) FAH.

<table>
<thead>
<tr>
<th>Catalogue</th>
<th>Q</th>
<th>FQ</th>
<th>SQ</th>
<th>SQ FAH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accommodation</td>
<td>186</td>
<td>112</td>
<td>73</td>
<td>94</td>
</tr>
<tr>
<td>Location</td>
<td>116</td>
<td>64</td>
<td>29</td>
<td>50</td>
</tr>
<tr>
<td>Event</td>
<td>92</td>
<td>57</td>
<td>39</td>
<td>52</td>
</tr>
<tr>
<td>Sport</td>
<td>102</td>
<td>49</td>
<td>31</td>
<td>46</td>
</tr>
<tr>
<td>Total</td>
<td>496</td>
<td>279</td>
<td>172</td>
<td>232</td>
</tr>
</tbody>
</table>

Figure 1: Interactive Query Management architecture

\[c: \text{a constraint to be relaxed}\]
\[\gamma \in [0, \infty]: \text{the expansion factor in a range constraint}\]
\[\delta \in [0, \infty]: \text{a parameter regulating the minimum relaxation in a range constraint}\]

\textbf{Return}: The relaxed version of \(c\)

\[
\text{relax}(c, \gamma, \delta)
\]
\[
\begin{align*}
1 & \quad \text{let } x \text{ be the feature constrained in } c \\
2 & \quad \text{if } c \text{ is not numeric then} \\
3 & \quad \quad \text{return } (x = *) \\
4 & \quad \text{end if} \\
5 & \quad \% c \text{ is numeric and } c = (x \in [l, u]) \\
6 & \quad \text{let } L \text{ and } U \text{ be the minimum and maximum values of } x \text{ in the catalogue} \\
7 & \quad \Delta \leftarrow \max\{\delta, \gamma(u - l)\} \\
8 & \quad \text{return } (x \in [l - \Delta, u + \Delta] \cap [L, U])
\end{align*}
\]

Figure 2: Single constraint relaxation
$q = \{c_1, \ldots, c_m\}$: a query to be relaxed.

$FAH = \{F^1, \ldots, F^k\}$: a set of feature abstraction hierarchies

**Return:** A suggestion list $SL = \{s_1, \ldots, s_k\}$, $k \leq m$, $s_i = (q_i, n_i)$, $q_i$ successful sub-query of $q$ and $n_i = \text{count}(q_i)$

---

RelaxQuery($q, FAH$)
1. $CL \leftarrow$ Partition constraints in $q$ according to $FAH$
2. $SL \leftarrow \emptyset$ % the list of relaxed-queries
3. $\text{gammas} \leftarrow (0.1, 0.1, \infty)$ % relaxation rates
4. $\delta \leftarrow 1$ % minimum relaxation for numeric constraints
5. **for each** $cl \in CL$ **do**
6. \hspace{1em} $q' \leftarrow q$
7. \hspace{1em} $s \leftarrow \emptyset$ % a suggestion
8. \hspace{1em} $i \leftarrow |cl|$
9. \hspace{2em} **while** $i > 0$ % starting from lowest abstract feature
10. \hspace{3em} $c \leftarrow$ get $i^{th}$ constraint from $cl$
11. \hspace{3em} $q' \leftarrow q' \setminus \{c\}$
12. \hspace{3em} $\text{relax} \leftarrow \text{true}$
13. \hspace{3em} $k \leftarrow 0$ % number of relaxation iterations
14. \hspace{3em} $n \leftarrow 0$ % the result size of the relaxed query
15. \hspace{3em} **while** ($\text{relax}$ and $k < \text{gammas.size}$) **do**
16. \hspace{4em} $c \leftarrow \text{relax}(c, \text{gammas}[k], \delta)$
17. \hspace{4em} $\text{relax} \leftarrow \text{false}$
18. \hspace{4em} $n \leftarrow \text{count}(q' \cup \{c\})$ % result size of the new query
19. \hspace{4em} **if** $n > 0$ **then**
20. \hspace{5em} $s \leftarrow (q' \cup \{c\}, n)$
21. \hspace{4em} **else if** $c$ is numeric **then**
22. \hspace{5em} $k \leftarrow k + 1$
23. \hspace{5em} $\text{relax} \leftarrow \text{true}$
24. \hspace{4em} **end if**
25. \hspace{4em} **end while**
26. \hspace{3em} **if** $s \neq \emptyset$ **then**
27. \hspace{4em} $SL \leftarrow SL \cup \{s\}$
28. \hspace{4em} $i \leftarrow 0$; % no need to go to higher abstr.
29. \hspace{3em} **else**
30. \hspace{4em} $i \leftarrow i - 1$
31. \hspace{3em} **end if**
32. **end for**
33. **return** $SL$

---

Figure 3: Query relaxation algorithm

---

32
$S$: A set of items - test set

$\lambda$: feature-selection method to be evaluated

$n_c$: number of constraints in the initial query

$p = (p_1, \ldots, p_n)$: user acceptance probabilities of features

$n_f$: number of features suggested by method $\lambda$ at each interaction

iterate: if true then the system shows $n_f$ additional features when none of the previous $n_f$ were accepted

FeatureSelectionEval($S, n_c, p, \lambda, n_f, iterate$)

1. for each $s \in S$ do
2.   $q \leftarrow \text{InitialQuery}(x, n_c)$ % build the initial query
3.   loop $\leftarrow$ true
4.   interaction $\leftarrow 0$
5.   features $\leftarrow$ true
6.   while count($q$) $> 50$ AND features do
7.     $sf \leftarrow \lambda(q)$ % sort features not yet constrained in $q$ with method $\lambda$
8.     $j \leftarrow 1$
9.     while loop AND $j \leq sf.size$ do
10.    $x_i \leftarrow sf_j$ % the $j$-th feature in $sf$
11.    if accept($x_i, p_i, s$) then
12.       $q \leftarrow q \cup \{x_i = s_i\}$
13.       interaction $\leftarrow$ interaction + 1
14.       loop $\leftarrow$ false % a new constraint has been found then exit while
15.    else
16.       if ($j$ modulo $n_f$ $= 0$) then
17.          interaction $\leftarrow$ interaction + 1
18.          if NOT iterate then
19.             features $\leftarrow$ false % not consider other $n_f$ features
20.          end if
21.       end if
22.    end if
23.    $j \leftarrow j + 1$
24.  end while
25.  if $j = sf.size + 1$ then $% q$ has not been updated
26.    features $\leftarrow$ false % exit from outer while
27.  end if
28. end while
29. end for

Figure 4: The feature selection evaluation procedure
Figure 5: The average result size of queries at each interaction (a), and (b) the average popularity of used features at each interaction.

Figure 6: Average result size at each interaction with a probabilistic acceptance model.
Figure 7: The average result size of queries at each interaction for two more data sets: (a) DTR-Destination catalogue, and (b) DTR-Lodging catalogue

Figure 8: RelaxQuery performance for different catalogues
Figure 9: Attempts needed to find a relaxation (k is the number of failing constraints).

Figure 10: NutKing graphical user interface for query tightening
Figure 11: NutKing graphical user interface for query relaxation

Figure 12: DieToRecs tightening user interface
Figure 13: DieToRecs relaxation user interface