Chapter 5

Knowledge-Based Recommender Systems

"*Knowledge is knowing that a tomato is a fruit. Wisdom is knowing not to put it in a fruit salad.*"–Brian O'Driscoll

5.1 Introduction

Both content-based and collaborative systems require a significant amount of data about past buying and rating experiences. For example, collaborative systems require a reasonably well populated ratings matrix to make future recommendations. In cases where the amount of available data is limited, the recommendations are either poor, or they lack full coverage over the entire spectrum of user-item combinations. This problem is also referred to as the *cold-start problem.* Different systems have varying levels of susceptibility to this problem. For example, collaborative systems are the most susceptible, and they cannot handle new items or new users very well. Content-based recommender systems are somewhat better at handling new items, but they still cannot provide recommendations to new users.

Furthermore, these methods are generally not well suited to domains in which the product is highly *customized.* Examples include items such as real estate, automobiles, tourism requests, financial services, or expensive luxury goods. Such items are bought rarely, and sufficient ratings are often not available. In many cases, the item domain may be complex, and there may be few instances of a specific item with a particular set of properties. For example, one might want to buy a house with a specific number of bedrooms, lawn, locality, and so on. Because of the complexity in describing the item, it may be difficult to obtain a reasonable set of ratings reflecting the past history of a user on a similar item. Similarly, an old rating on a car with a specific set of options may not even be relevant in the present context.

C.C. Aggarwal, *Recommender Systems: The Textbook*,
DOI 10.1007/978-3-319-29659-3_5

How can one handle such customization and paucity of ratings? Knowledge-based recommender systems rely on *explicitly soliciting user requirements* for such items. However, in such complex domains, it is often difficult for users to fully enunciate or even understand how their requirements match the product availability. For example, a user may not even be aware that a car with a certain combination of fuel efficiency and horsepower is available. Therefore, such systems use interactive feedback, which allows the user to explore the inherently complex product space and learn about the trade-offs available between various options. The retrieval and exploration process is facilitated by knowledge bases describing the utilities and/or trade-offs of various features in the product domain. The use of knowledge bases is so important to an effective retrieval and exploration process, that such systems are referred to as knowledge-based recommender systems.

Knowledge-based recommender systems are well suited to the recommendation of items that are not bought on a regular basis. Furthermore, in such item domains, users are generally more active in being explicit about their requirements. A user may often be willing to accept a movie recommendation without much input, but she would be unwilling to accept recommendations about a house or a car without having detailed information about the specific features of the item. Therefore, knowledge-based recommender systems are suited to types of item domains different from those of collaborative and content-based systems. In general, knowledge-based recommender systems are appropriate in the following situations:

1. Customers want to explicitly specify their requirements. Therefore, interactivity is a crucial component of such systems. Note that collaborative and content-based systems do not allow this type of detailed feedback.

2. It is difficult to obtain ratings for a specific type of item because of the greater complexity of the product domain in terms of the types of items and options available.

3. In some domains, such as computers, the ratings may be time-sensitive. The ratings on an old car or computer are not very useful for recommendations because they evolve with changing product availability and corresponding user requirements.

A crucial part of knowledge-based systems is the greater control that the user has in guiding the recommendation process. This greater control is a direct result of the need to be able to specify detailed requirements in an inherently complex problem domain. At a basic level, the conceptual differences in the three categories of recommendations are described in Table 5.1. Note that there are also significant differences in the input data used by various systems. The recommendations of content-based and collaborative systems are primarily based on *historical* data, whereas knowledge-based systems are based on the direct specifications by users of *what they want*. An important distinguishing characteristic of knowledge-based systems is a high level of *customization* to the specific domain. This customization is achieved through the use of a knowledge-base that encodes relevant *domain knowledge* in the form of either constraints or similarity metrics. Some knowledge-based systems might also use user attributes (e.g., demographic attributes) in addition to item attributes, which are specified at query time. In such cases, the domain knowledge might also encode relationships between user attributes and item attributes. The use of such attributes is, however, not universal to knowledge-based systems, in which the greater focus is on user *requirements*.

Knowledge-based recommender systems can be categorized on the basis of user interactive methodology and the corresponding knowledge bases used to facilitate the interaction. There are two primary types of knowledge-based recommender systems:

1. *Constraint-based recommender systems:* In constraint-based systems [196, 197], users typically specify requirements or constraints (e.g., lower or upper limits) on the item

Table 5.1: The conceptual goals of various recommender systems

Approach	**Conceptual Goal**	**Input**
Collaborative	Give me recommendations based on a collaborative approach that leverages the ratings and actions of my peers/myself.	User ratings + community ratings
Content-based	Give me recommendations based on the content (attributes) I have favored in my past ratings and actions.	User ratings + item attributes
Knowledge-based	Give me recommendations based on my explicit specification of the kind of content (attributes) I want.	User specification + item attributes + domain knowledge

attributes. Furthermore, domain-specific rules are used to match the user requirements or attributes to item attributes. These rules represent the domain-specific knowledge used by the system. Such rules could take the form of domain-specific constraints on the item attributes (e.g., "*Cars before year 1970 do not have cruise control.*"). Furthermore, constraint-based systems often create rules relating user attributes to item attributes (e.g., "*Older investors do not invest in ultrahigh-risk products.*"). In such cases, user attributes may also be specified in the search process. Depending on the number and type of returned results, the user might have an opportunity to modify their original requirements. For example, a user might relax some constraints when too few results are returned, or add more constraints when too many results are returned. This search process is interactively repeated until the user arrives at her desired results.

2. *Case-based recommender systems:* In case-based recommender systems [102, 116, 377, 558], specific cases are specified by the user as targets or anchor points. Similarity metrics are defined on the item attributes to retrieve similar items to these targets. The similarity metrics are often carefully defined in a domain-specific way. Therefore, the similarity metrics form the domain knowledge that is used in such systems. The returned results are often used as new target cases with some interactive modifications by the user. For example, when a user sees a returned result that is almost similar to what she wants, she might re-issue a query with that target, but with some of the attributes changed to her liking. Alternatively, a *directional critique* may be specified to prune items with specific attribute values greater (or less) than that of a specific item of interest. This interactive process is used to guide the user towards the final recommendation.

Note that in both cases, the system provides an opportunity for the user to change her specified requirements. However, the way in which this is done is different in the two cases. In case-based systems, examples (or *cases*) are used as anchor points to guide the search in conjunction with *similarity metrics*, whereas in constraint-based systems, specific criteria/rules (or *constraints*) are used to guide the search. In both cases, the presented results are used to modify the criteria for finding further recommendations. Knowledge-based systems derive their name from the fact that they encode various types of *domain knowledge* in the form of constraints, rules, similarity metrics, and utility functions during the search process. For example, the design of a similarity metric or a specific constraint requires domain-specific knowledge, which is crucial to the effective functioning of the recommender system. In general, knowledge-based systems draw on highly heterogeneous, domain-specific sources of knowledge, compared to content-based and collaborative systems, which work with somewhat similar types of input data across various domains. As a result, knowledge-based systems are highly customized, and they are not easily generalizable across various domains. However, the broader principles with which this customization is done are invariant across domains. The goal of this chapter is to discuss these principles.

The interaction between user and recommender may take the form of *conversational systems*, *search-based systems*, or *navigational systems*. Such different forms of guidance may be present either in isolation, or in combination, and they are defined as follows:

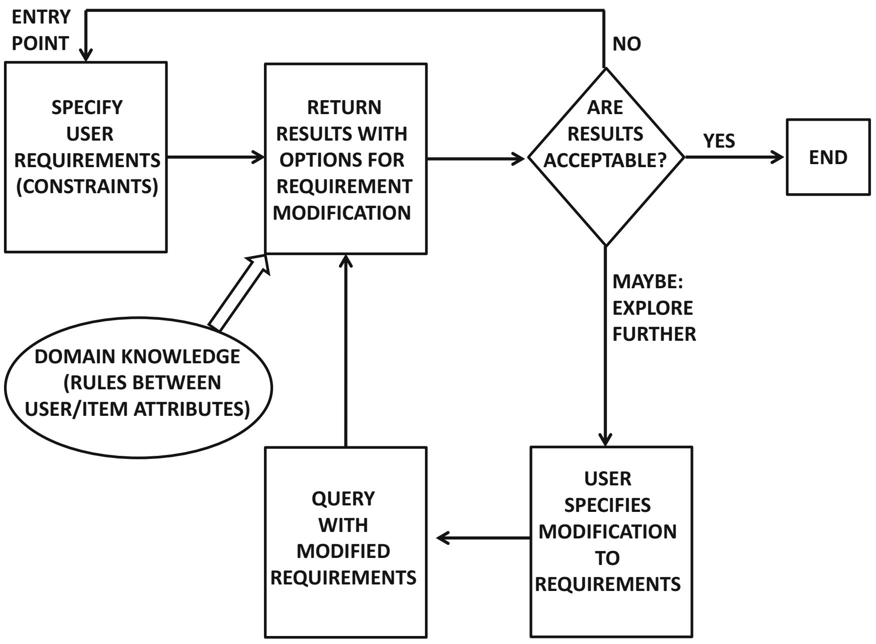
1. *Conversational systems:* In this case, the user preferences are determined in the context of a feedback loop. The main reason for this is that the item domain is complex, and the user preferences can be determined only in the context of an iterative conversational system.

2. *Search-based systems:* In search-based systems, user preferences are elicited by using a preset sequence of questions such as the following: "Do you prefer a house in a suburban area or within the city?"

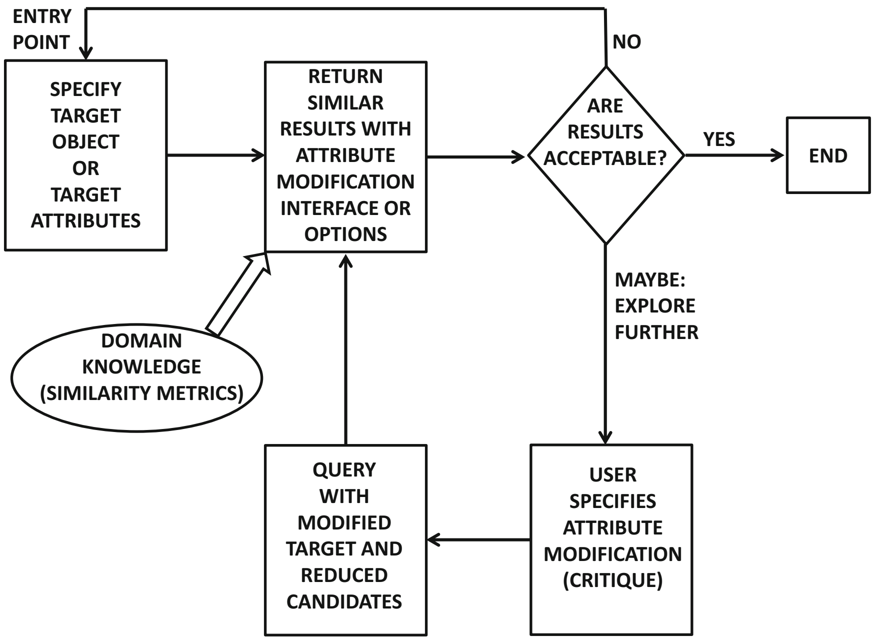
3. *Navigation-based recommendation:* In navigation-based recommendation, the user specifies a number of change requests to the item being currently recommended. Through an iterative set of change requests, it is possible to arrive at a desirable item. An example of a change request specified by the user, when a specific house is being recommended is as follows: "I would like a similar house about 5 miles west of the currently recommended house." Such recommender systems are also referred to as *critiquing recommender systems* [120, 121, 417].

These different forms of guidance are well suited to different types of recommender systems. For example, critiquing systems are naturally designed for case-based recommenders, because one critiques a specific case in order to arrive at the desired outcome. On the other hand, a search-based system can be used to set up user requirements for constraint-based recommenders. Some forms of guidance can be used with both constraint-based and case-based systems. Furthermore, different forms of guidance can also be used in combination in a knowledge-based system. There are no strict rules as to how one might design the interface for a knowledge-based system. The goal is always to guide the user through a complex product space.

Typical examples of the interactive process in constraint-based recommenders and case-based recommenders are illustrated in Figures 5.1(a) and (b), respectively. The overall interactive approach is quite similar. The main difference in the two cases is in terms of how the user specifies the queries and interacts with the system for subsequent refinement. In constraint-based systems, specific requirements (or *constraints*) are specified by the user, whereas in case-based systems, specific targets (or *cases*) are specified. Correspondingly, different types of interactive processes and domain knowledge are used in the two systems. In constraint-based systems, the original query is modified by addition, deletion, modification, or relaxation of the original set of user requirements. In case-based systems, either the target is modified through user interaction, or the search results are pruned through the use of *directional* critiques. In such critiques, the user simply states whether a specific attribute in the search results needs to be increased, decreased, or changed in a certain way. Such an approach represents a more conversational style than simply modifying the target. In both these types of systems, a common motivation is that users are often not in a position to exactly state their requirements up front in a complex product domain. In constraint-based systems, this problem is partially addressed through a knowledge-base of rules, which map user requirements to product attributes. In case-based systems, this problem is addressed


ENTRY POINT
SPECIFY USER REQUIREMENTS (CONSTRAINTS)
RETURN RESULTS WITH OPTIONS FOR REQUIREMENT MODIFICATION
ARE RESULTS ACCEPTABLE?
NO
YES
END
MAYBE: EXPLORE FURTHER
DOMAIN KNOWLEDGE (RULES BETWEEN USER/ITEM ATTRIBUTES)
QUERY WITH MODIFIED REQUIREMENTS
USER SPECIFIES MODIFICATION TO REQUIREMENTS

(a) Constraint-based interaction


ENTRY POINT
SPECIFY TARGET OBJECT OR TARGET ATTRIBUTES
RETURN SIMILAR RESULTS WITH ATTRIBUTE MODIFICATION INTERFACE OR OPTIONS
ARE RESULTS ACCEPTABLE?
NO
YES
END
MAYBE: EXPLORE FURTHER
DOMAIN KNOWLEDGE (SIMILARITY METRICS)
QUERY WITH MODIFIED TARGET AND REDUCED CANDIDATES
USER SPECIFIES ATTRIBUTE MODIFICATION (CRITIQUE)

(b) Case-based interaction

Figure 5.1: Overview of interactive process in knowledge-based recommenders

Table 5.2: Examples of attributes in a recommendation application for buying homes

Item-Id	Beds.	Baths.	Locality	Type	Floor Area	Price
1	3	2	Bronx	Townhouse	1600	220,000
2	5	2.5	Chappaqua	Split-level	3600	973,000
3	4	2	Yorktown	Ranch	2600	630,000
4	2	1.5	Yorktown	Condo	1500	220,000
5	4	2	Ossining	Colonial	2700	430,000

through a conversational style of critiquing. The interactive aspect is common to both systems, and it is crucial in helping the users discover how the items in a complex product domain fit their needs.

It is noteworthy that most forms of knowledge-based recommender systems depend heavily on the descriptions of the items in the form of relational attributes rather than treating them as text keywords like[1] content-based systems. This is a natural consequence of the inherent complexity in knowledge-based recommendations in which domain-specific knowledge can be more easily encoded with relational attributes. For example, the attributes for a set of houses in a real-estate application is illustrated in Table 5.2. In case-based recommenders, the similarity metrics are defined in terms of these attributes in order to provide similar matches to target homes provided by the user. Note that each relational attribute would have a different significance and weight in the matching process, depending on domain-specific criteria. In constraint-based systems, the queries are specified in the form of requirements on these attributes, such as a maximum price on the house, or a specific locality. Therefore, the problem reduces to an instance of the *constraint-satisfaction problem*, where one must identify the relevant set of instances satisfying all the constraints.

This chapter is organized as follows. Constraint-based recommenders are introduced in section 5.2. Case-based recommenders are discussed in section 5.3. The use of persistent personalization in knowledge-based systems is discussed in section 5.4. A summary is given in section 5.5.

5.2 Constraint-Based Recommender Systems

Constraint-based recommender systems allow the users to specify hard requirements or constraints on the item attributes. Furthermore, a set of rules is used in order to match the customer requirements with item attributes. However, the customers may not always specify their queries in terms of the same attributes that describe the items. Therefore, an additional set of rules is required that relates the customer requirements with the product attributes. In relation to the previous home-buying example in Table 5.2, some examples of customer-specified attributes are as follows:

Marital-status (categorical), *Family-Size* (numerical), *suburban-or-city* (binary), *Min-Bedrooms* (numerical), *Max-Bedrooms* (numerical), *Max-Price* (numerical)

These attributes may represent either inherent customer properties (e.g., demographics), or they may specify customer requirements for the product. Such requirements are usually

[1]Content-based systems are used both in the information retrieval and the relational settings, whereas knowledge-based systems are used mostly in the relational setting.

specified interactively during the dialog between the customer and the recommender system. Note that many of the requirement attributes are not included in Table 5.2. While the mappings of some of the customer requirement attributes, such as *Max-Price*, to product attributes are obvious, the mappings of others, such as *suburban-or-rural*, are not quite as obvious. Similarly, in a financial application, a customer may specify a product requirement such as "*conservative investments*," which needs to be mapped to concrete product attributes (e.g., *Asset-type*=*Treasuries*) directly describing the products. Clearly, one must somehow be able to map these customer attributes/requirements into the product attributes in order to filter products for recommendation. This is achieved through the use of knowledge bases. The knowledge bases contain additional rules that map customer attributes/requirements to the product attributes:

$$\textit{Suburban-or-rural}{=}\textit{Suburban} \Rightarrow \textit{Locality}{=}\ \langle \textit{List of relevant localities}\ \rangle$$

Such rules are referred to as *filter* conditions because they map user requirements to the item attributes and use this mapping to filter the retrieved results. Note that these types of rules may be either derived from the product domain, or, more rarely, they may be derived by historical mining of such data sets. In this particular case, it is evident that this rule can be derived directly using publicly available geographical information. Another example is the car domain, where certain optional packages may be valid only with certain other attributes. For example, a high-torque engine may be available only in a sports model. Such conditions are also referred to as *compatibility* conditions, because they can be used to quickly discover inconsistencies in the user-specified requirements with the product domain. In many cases, such compatibility constraints can be integrated within the user interface. For example, the car pricing site Edmunds.com prevents users from entering mutually inconsistent requirements within the user interface. In other cases, where inconsistency detection is not possible within the user interface, such inconsistencies can be detected at query processing time by returning empty sets of results.

Some of the other compatibility constraints may relate customer attributes to one another. Such constraints are useful when customers specify personal information (e.g., demographic information) about themselves during the interactive session. For example, demographic attributes may be related to customer product requirements based on either domain-specific constraints, or historical experience. An example of such a constraint is as follows:

$$\textit{Marital-status}{=}\textit{single} \Rightarrow \textit{Min-Bedrooms}{\leq}\textit{5}$$

Presumably, by either domain-specific experience or through data mining of historical data sets, it has been inferred that single individuals do not prefer to buy very large houses. Similarly, a small home might not be suitable for a very large family. This constraint is modeled with the following rule:

$$\textit{Family-Size}{\geq}\textit{5} \Rightarrow \textit{Min-Bedrooms}{\geq}\textit{3}$$

Thus, there are three primary types of input to the constraint-based recommender system:

1. The first class of inputs is represented by the attributes describing the inherent properties of the user (e.g., demographics, risk profiles) and specific requirements in the product (e.g., *Min-Bedrooms*). Some of these attributes are easy to relate to product attributes, whereas others can be related to product attributes only through the use of knowledge bases. In most cases, the customer properties and requirements are specified interactively in a session, and they are not persistent across multiple sessions.

Therefore, if another user specifies the same set of requirements in a session, they will obtain the same result. This is different from other types of recommender systems, where the personalization is persistent because it is based on historical data.

2. The second class of inputs is represented by knowledge bases, which map customer attributes/requirements to various product attributes. The mapping can be achieved either directly or indirectly as follows:

    - **Directly:** These rules relate customer requirements to hard requirements on product attributes. An example of such a rule is as follows:

      $$\textit{Suburban-or-rural=Suburban} \Rightarrow \textit{Locality=} \langle \textit{List of relevant localities} \rangle$$
      $$\textit{Min-Bedrooms} \geq 3 \Rightarrow \textit{Price} \geq 100{,}000$$

      Such rules are also referred to as filter conditions.

    - **Indirectly:** These rules relate customer attributes/requirements to typically expected product requirements. Therefore, such rules can also be viewed as an indirect way of relating customer attributes to product attributes. Examples of such rules are as follows:

      $$\textit{Family-Size} \geq 5 \Rightarrow \textit{Min-Bedrooms} \geq 3$$
      $$\textit{Family-Size} \geq 5 \Rightarrow \textit{Min-Bathrooms} \geq 2$$

      Note that the conditions on both sides of the rule represent customer attributes, although the ones on the right-hand side are generally customer requirements, which can be mapped to product attributes easily. These constraints represent compatibility constraints. In the event that the compatibility constraints or filter conditions are inconsistent with the customer-specified requirements, the recommended list of items will be empty.

    The aforementioned knowledge bases are derived from publicly available information, domain experts, past experience, or data mining of historical data sets. Therefore, a significant amount of effort is involved in building the knowledge bases.

3. Finally, the product *catalog* contains a list of all the products together with the corresponding item attributes. A snapshot of a product catalog for the home-buying example is illustrated in Table 5.2.

Therefore, the problem boils down to determining all the instances in the available product list that satisfy the customer requirements and the rules in the knowledge base.

5.2.1 Returning Relevant Results

The problem of returning relevant results can be shown to be an instance of the constraint satisfaction problem by viewing each item in the catalog as a constraint on the attributes and expressing the catalog in disjunctive normal form. This expression is then combined with the rules in the knowledge base to determine whether a mutually consistent region of the product space exists.

More simply, the set of rules and requirements can be reduced to a data filtering task on the catalog. All the customer requirements and the active rules relevant to the customer are used to construct a database selection query. The steps for creating such a filtering query are as follows:

1. For each requirement (or personal attribute) specified by the customer in their user interface, it is checked whether it matches the antecedent of a rule in the knowledge base. If such a matching exists, then the consequent of that rule is treated as a valid selection condition. For example, consider the aforementioned real-estate example. If the customer has specified *Family-Size=6* and *ZIP Code=10547* among their personal attributes and preferences in the user interface, then it is detected that *Family-Size*=6 triggers the following rules:

   $$\textit{Family-Size} \geq 5 \Rightarrow \textit{Min-Bedrooms} \geq 3$$
   $$\textit{Family-Size} \geq 5 \Rightarrow \textit{Min-Bathrooms} \geq 2$$

   Therefore, the consequents of these conditions are added to the user requirements. The rule base is again checked with these expanded requirements, and it is noticed that the newly added constraint *Min-Bedrooms*≥ 3 triggers the following rules:

   $$\textit{Min-Bedrooms} \geq 3 \Rightarrow \textit{Price} \geq 100{,}000$$
   $$\textit{Min-Bedrooms} \geq 3 \Rightarrow \textit{Bedrooms} \geq 3$$
   $$\textit{Min-Bathrooms} \geq 3 \Rightarrow \textit{Bathrooms} \geq 2$$

   Therefore, the conditions *Price≥100,000*, and the range constraints on the requirement attributes *Min-Bedrooms* and *Min-Bathrooms* are replaced with those on the product attributes *Bedrooms* and *Bathrooms*. In the next iteration, it is found that no further conditions can be added to the user requirements.

2. These expanded requirements are used to construct a database query in conjunctive normal form. This represents a traditional database selection query, which computes the intersection of the following constraints on the product catalog:

   $$(\textit{Bedrooms} \geq 3) \wedge (\textit{Bathrooms} \geq 2) \wedge (\textit{Price} \geq 100{,}000) \wedge (\textit{ZIP Code} = 10547)$$

   Note that the approach essentially maps all customer attribute constraints and requirement attribute constraints to constraints in the product domain.

3. This selection query is then used to retrieve the instances in the catalog that are relevant to the user requirements.

It is noteworthy that most constraint-based systems enable specification of all user requirements or other attributes (e.g., preferences, demographic information) *during the session itself*. In other words, the specified information is typically not persistent; if a different user specifies the same input, they will get exactly the same result. This characteristic is common to most knowledge-based systems. Section 5.4 will discuss some recent advancements in the *persistent* personalization of knowledge-based systems.

The resulting list of items, which satisfy the constraints, is then presented to the user. The methodology for ranking the items is discussed later in this section. The user may then modify her requirements further to obtain more refined recommendations. The overall process of exploration and refinement often leads the customer to discover recommendations that she might otherwise not have been able to arrive at on her own.

5.2.2 Interaction Approach

The interaction between the user and the recommender system generally proceeds in three phases.

1. An interactive interface is used by the user to specify her initial preferences. A common approach is to use a Web style form in which the desired values of the attributes may be entered. An example of a *hypothetical* interface for home buying, which we will be using as a running example, is provided in Figure 5.2. Alternatively, the user could be asked a series of questions to elicit her initial preferences. For example, the car recommendation site Edmunds.com presents a series of interfaces to the users to specify their preferences about the specific features they might want. The answers to the queries in the first interface may affect the questions in the next interface.

2. The user is presented with a ranked list of matching items. An explanation for why the items are returned is typically provided. In some cases, no items might match the user requirements. In such cases, possible relaxations of the requirements might be suggested. For example, in Figure 5.3, no results are returned by the query, and possible relaxations are suggested. In cases, where too many items are returned, suggestions for possible constraints (user requirements) are included. For example, in Figure 5.4, too many results are returned. Possible constraints are suggested to be added to the query.

3. The user then refines her requirements depending on the returned results. This refinement might take the form of the addition of further requirements, or the removal of some of the requirements. For example, when an empty set is returned, it is evident that some of the requirements need to be relaxed. Constraint satisfaction methods are used to identify possible sets of candidate constraints, which might need to be relaxed. Therefore, the system generally helps the user in making her modifications in a more intelligent and efficient way.

Thus, the overall approach uses an iterative feedback loop to assist the users in making meaningful decisions. It is crucial to design a system that can guide the user towards requirements that increase her awareness regarding the available choices.

There are several aspects of this interaction, in which explicit computation is required in order to help the user. For example, a user will typically not be able specify desired values for all the product attributes. For instance, in our home-buying example, the user may specify constraints only on the number of bedrooms and not specify any constraints on the price. Several solutions are possible under this scenario:

1. The system may leave the other attributes unconstrained and retrieve the results based on only the specified constraints. For example, all possible ranges of prices may be considered in order to provide the first set of responses to the user. Although this may be the most reasonable choice, when the user query has been formulated well, it may not be an effective solution in cases where the number of responses is large.

2. In some cases, default values may be suggested to the user to provide guidance. The default values can be used only to guide the user in selecting values, or they can actually be included in the query if the user does not select any value (including the default) for that attribute. It can be argued that including a default value within the query (without explicit specification) can lead to significant bias within the recommender system, especially when the defaults are not very well researched. In general,

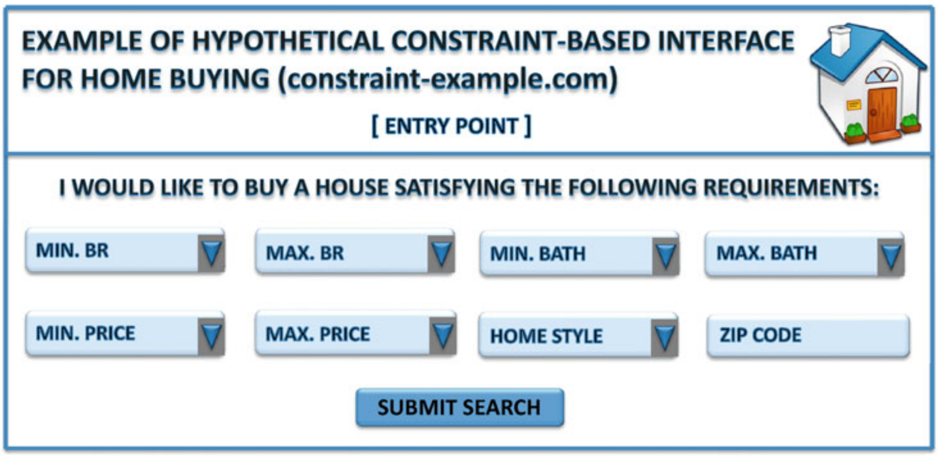
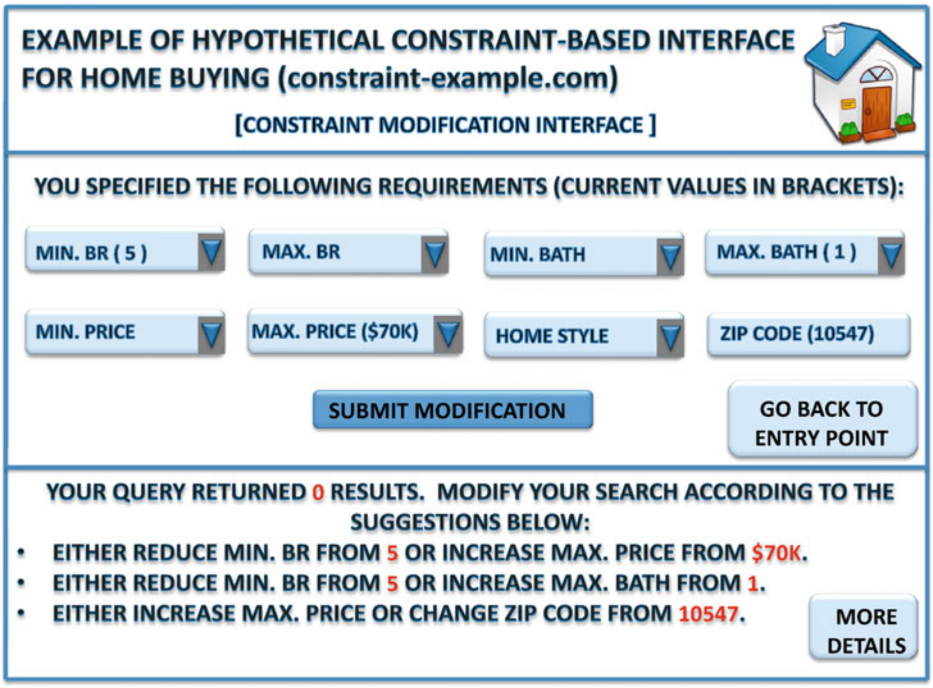

EXAMPLE OF HYPOTHETICAL CONSTRAINT-BASED INTERFACE FOR HOME BUYING (constraint-example.com)
[ENTRY POINT]
I WOULD LIKE TO BUY A HOUSE SATISFYING THE FOLLOWING REQUIREMENTS:
MIN. BR
MAX. BR
MIN. BATH
MAX. BATH
MIN. PRICE
MAX. PRICE
HOME STYLE
ZIP CODE
SUBMIT SEARCH

Figure 5.2: A hypothetical example of an initial user interface for a constraint-based recommender (`constraint-example.com`)

> default values should be used only as a suggestion for the user. This is because the main goal of defaults should be to *guide* the user towards natural values, rather than to *substitute* for unspecified options.

How are default values determined? In most cases, it is necessary to choose the defaults in a domain-specific way. Furthermore, some values of the defaults may be affected by others. For example, the horsepower of a selected car model might often reflect the desired fuel efficiency. Knowledge bases need to explicitly store the data about such default values. In some cases, where the historical data from user sessions is available, it is possible to learn the default values. For the various users, their specified attribute values in the query sessions may be available, including the missing values. The average values across various sessions may be used as defaults. Consider a query session initiated by Alice for buying cars. Initially, her defaults are computed on the basis of the average values in historical sessions. However, if she specifies the desired horsepower of the car, then the interface automatically adjusts her default value of the fuel efficiency. This new default value is based on the average of fuel efficiency of cars, which were specified in historical sessions for cars with similar horsepower. In some cases, the system might automatically adjust the default values based on feasibility constraints with respect to the knowledge base. As users specify increasingly more values in the interface, the average can be computed only over the sessions within the neighborhood of the current specification.

After the query has been issued, the system provides a ranked list of possible matches from the catalog. Therefore, it is important to be able to meaningfully rank the matches and also provide explanations for the recommended results if needed. In cases, where the returned set of matches is too small or too large, further guidance may be provided to the user on either relaxing or tightening requirements. It is noteworthy that the provision of explanations is also an intelligent way of guiding the user towards more meaningful query refinements. In the following, we will discuss these various aspects of interactive user guidance.


EXAMPLE OF HYPOTHETICAL CONSTRAINT-BASED INTERFACE FOR HOME BUYING (constraint-example.com)

[CONSTRAINT MODIFICATION INTERFACE]

YOU SPECIFIED THE FOLLOWING REQUIREMENTS (CURRENT VALUES IN BRACKETS):

MIN. BR (5)	MAX. BR	MIN. BATH	MAX. BATH (1)
MIN. PRICE	MAX. PRICE ($70K)	HOME STYLE	ZIP CODE (10547)

YOUR QUERY RETURNED 0 RESULTS. MODIFY YOUR SEARCH ACCORDING TO THE SUGGESTIONS BELOW:

- EITHER REDUCE MIN. BR FROM 5 OR INCREASE MAX. PRICE FROM $70K.
- EITHER REDUCE MIN. BR FROM 5 OR INCREASE MAX. BATH FROM 1.
- EITHER INCREASE MAX. PRICE OR CHANGE ZIP CODE FROM 10547.

Figure 5.3: A hypothetical example of a user interface for handling empty query results in a constraint-based recommender (`constraint-example.com`)

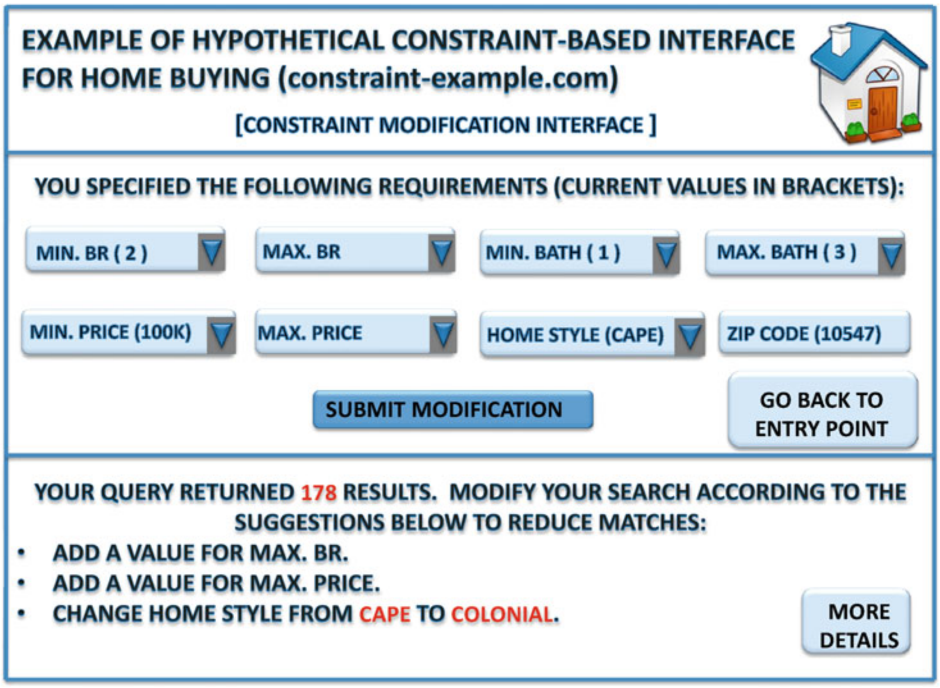
5.2.3 Ranking the Matched Items

A number of natural methods exist for ranking the items according to user requirements. The simplest approach is to allow the user to specify a single numerical attribute on the basis of which to rank the items. For example, in the home-buying application, the system might provide the user the option to rank the items on the basis of (any one of) the home price, number of bedrooms, or distance from a particular ZIP code. This approach is, in fact, used in many commercial interfaces.

Using a single attribute has the drawback that the importance of other attributes is discounted. A common approach is to use utility functions in order to rank the matched items. Let $\overline{V} = (v_1 \ldots v_d)$ be the vector of values defining the attributes of the matched products. Therefore, the dimensionality of the content space is d. The utility functions may be defined as weighted functions of the utilities of individual attributes. Each attribute has a weight w_j assigned to it, and it has a contribution defined by the function $f_j(v_j)$ depending on the value v_j of the matched attribute. Then, the utility $U(\overline{V})$ of the matched item is given by the following:

$$U(\overline{V}) = \sum_{j=1}^{d} w_j \cdot f_j(v_j) \tag{5.1}$$

Clearly, one needs to instantiate the values of w_j and $f_j(\cdot)$ in order to learn the utility function. The design of effective utility functions often requires domain-specific knowledge, or learning data from past user interactions. For example, when v_j is numeric, one might assume that the function $f_j(v_j)$ is linear in v_j, and then learn the coefficients of the linear


EXAMPLE OF HYPOTHETICAL CONSTRAINT-BASED INTERFACE FOR HOME BUYING (constraint-example.com)

[CONSTRAINT MODIFICATION INTERFACE]

YOU SPECIFIED THE FOLLOWING REQUIREMENTS (CURRENT VALUES IN BRACKETS):

YOUR QUERY RETURNED 178 RESULTS. MODIFY YOUR SEARCH ACCORDING TO THE SUGGESTIONS BELOW TO REDUCE MATCHES:

- ADD A VALUE FOR MAX. BR.
- ADD A VALUE FOR MAX. PRICE.
- CHANGE HOME STYLE FROM CAPE TO COLONIAL.

Figure 5.4: A hypothetical example of a user interface for handling too many query results in a constraint-based recommender (`constraint-example.com`)

function as well as w_j by eliciting feedback from various users. Typically, training data is elicited from some users who are given the task of ranking some sample items. These ranks are then used to learn the aforementioned model with the use of regression models. This approach is related to the methodology of *conjoint analysis* [155, 531]. Conjoint analysis defines statistical methods for the formal study of how people value the different attributes that make up an individual product or service. The bibliographic notes contain pointers to some methods that are commonly used for the design of utility functions.

5.2.4 Handling Unacceptable Results or Empty Sets

In many cases, a particular query might return an empty set of results. In other cases, the set of returned results might not be large enough to meet the user requirements. In such cases, a user has two options. If it is deemed that a straightforward way of repairing the constraints does not exist, she may choose to start over from the entry point. Alternatively, she may decide to change or relax the constraints for the next interactive iteration.

How can the user make a meaningful choice on whether to relax the constraints and in what way? In such cases, it is often helpful to provide the user with some guidance on relaxing the current requirements. Such proposals are referred to as *repair proposals*. The idea is to be able to determine minimal sets of inconsistent constraints, and present them to the user. It is easier for the user to assimilate minimal sets of inconsistent constraints, and find ways of relaxing one or more of the constraints in these sets. Consider the home-buying example, in which it may be found that the user has specified many requirements, but the only mutually inconsistent pair of requirements is *Max-Price* $< 100,000$ and *Min-Bedrooms* > 5.

If this pair of constraints is presented to the user, she can understand that she either needs to increase the maximum price she is willing to pay, or she needs to settle for a smaller number of bedrooms. A naive way of finding the minimal set of inconsistent constraints is to perform a bottom-up search of all combinations of user requirements, and determine the smallest sets that are infeasible. In many interactive interfaces, the user might specify only a small number of (say, 5 to 10) requirements, and the number of constraints involving these attributes (in the domain knowledge) might also be small. In such cases, exhaustive exploration of all the possibilities is not an unreasonable approach. By its very nature, interactive requirement specification often results in the specification of a relatively small number of constraints. It is unusual for a user to specify 100 different requirements in an interactive query. In some cases, however, when the number of user-specified requirements is large and the domain knowledge is significant, such an exhaustive bottom-up exploration might not be a feasible option. More sophisticated methods, such as *QUICKXPLAIN* and *MINRELAX*, have also been proposed, which can be used for fast discovery of small conflicting sets and minimal relaxations [198, 273, 274, 289, 419].

Most of these methods use similar principles; small sets of violating constraints are determined, and the most appropriate relaxations are suggested based on some pre-defined criteria. In real applications, however, it is sometimes difficult to suggest concrete criteria for constraint relaxation. Therefore, a simple alternative is to present the user with small sets of inconsistent constraints, which can often provide sufficient intuition to the user in formulating modified constraints.

5.2.5 Adding Constraints

In some cases, the number of returned results may be very large, and the user may need to suggest possible constraints to be added to the query. In such cases, a variety of methods can be used to suggest constraints to the user along with possible default values. The attributes for such constraints are often chosen by mining historical session logs. The historical session logs can either be defined over all users, or over the particular user at hand. The latter provides more personalized results, but may often be unavailable for infrequently bought items (e.g., cars or houses). It is noteworthy that knowledge-based systems are generally designed to not use such persistent and historical information precisely because they are designed to work in cold-start settings; nevertheless, such information can often be very useful in improving the user experience when it is available.

How can historical session data be used? The idea is to select constraints that are popular. For example, if a user has specified the constraints on a set of item attributes, then other sessions containing one or more of these attributes are identified. For example, if a user has specified constraints on the number of bedrooms and the price, previous sessions containing constraints on the bedroom and price are identified. In particular, the top-k nearest neighbor sessions in terms of the number of common attributes are identified. If it is determined that the most popular constraint among these top-k sessions is on the number of bathrooms, then this attribute is suggested by the interface as a candidate for adding additional constraints.

In many cases, the temporal ordering in which users have specified constraints in the past is available. In such cases, it is also possible to use the *order* in which the customer specified the constraints by treating the constraints as an ordered set, rather than as an unordered set [389]. A simple way of achieving this goal is to determine the most frequent attribute that *follows* the current specified set of constrained attributes in previous sessions. Sequential pattern mining can be used to determine such frequent attributes. The works in [389, 390] model the sequential learning problem as a Markov Decision Process (MDP), and use reinforcement learning techniques to measure the impact of various choices. The constraints can be suggested based on their selectivity in the database or based on the average specification of the user in past sessions.

5.3 Case-Based Recommenders

In case-based recommenders, similarity metrics are used to retrieve examples that are similar to the specified targets (or *cases*). For instance, in the real-estate example of Table-5.2, the user might specify a locality, the number of bedrooms, and a desired price to specify a target set of attributes. Unlike constraint-based systems, no *hard* constraints (e.g., minimum or maximum values) are enforced on these attributes. It is also possible to design an initial query interface in which examples of relevant items are used as targets. However, it is more natural to specify desired properties in the initial query interface. A similarity function is used to retrieve the examples that are most similar to the user-specified target. For example, if no homes are found specifying the user requirements exactly, then the similarity function is used to retrieve and rank items that are as similar as possible to the user query. Therefore, unlike constraint-based recommenders, the problem of retrieving empty sets is not an issue in case-based recommenders.

There are also substantial differences between a constraint-based recommender and a case-based recommender in terms of how the results are refined. Constraint-based systems use requirement relaxation, modification, and tightening to refine the results. The earliest case-based systems advocated the repeated modification of user query requirements until a suitable solution could be found. Subsequently, the method of *critiquing* was developed. The general idea of critiquing is that users can select one or more of the retrieved results and specify further queries of the following form:

"*Give me more items like X, but they are different in attribute(s) Y according to guidance Z.*"

A significant variation exists in terms of whether one or more than one attributes is selected for modification and how the guidance for modifying the attributes is specified. The main goal of critiquing is to support interactive browsing of the item space, where the user gradually becomes aware of further options available to them through the retrieved examples. Interactive browsing of the item space has the advantage that it is a learning process for the user during the process of iterative query formulation. It is often possible that through repeated and interactive exploration, the user might be able to arrive at items that could not otherwise have been reached at the very beginning.

For example, consider the home-buying example of Table 5.2. The user might have initially specified a desired price, the number of bedrooms, and a desired locality. Alternatively, the user might specify a target address to provide an example of a possible house she might be interested in. An example of an initial interface in which the user can specify the target in two different ways, is illustrated in Figure 5.5. The top portion of the interface illustrates the specification of target features, whereas the bottom portion of the interface illustrates the specification of a target address. The latter approach is helpful in domains where the users have greater difficulty in specifying technically cryptic features. An example might be the case of digital cameras, where it is harder to specify all the technical features exactly for a non-specialist in photography. Therefore, a user might specify her friend's camera as

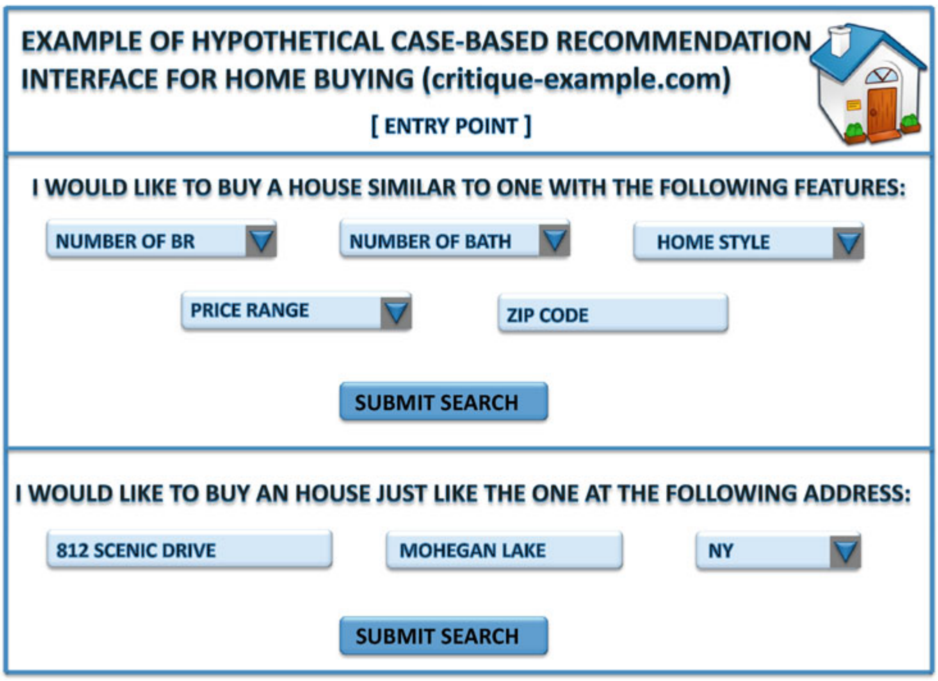

EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)
[ENTRY POINT]
I WOULD LIKE TO BUY A HOUSE SIMILAR TO ONE WITH THE FOLLOWING FEATURES:
NUMBER OF BR
NUMBER OF BATH
HOME STYLE
PRICE RANGE
ZIP CODE
SUBMIT SEARCH
I WOULD LIKE TO BUY AN HOUSE JUST LIKE THE ONE AT THE FOLLOWING ADDRESS:
812 SCENIC DRIVE
MOHEGAN LAKE
NY
SUBMIT SEARCH

Figure 5.5: A hypothetical example of an initial user interface in a case-based recommender (`critique-example.com`)

the target case, rather than specifying all the technical features. Note that this interface is hypothetically designed for illustrative purposes only, and it is not based on an actual recommender system.

The system uses the target query in conjunction with similarity or utility functions in order to retrieve matching results. Eventually, upon retrieving the results, the user might decide to like a particular house, except that its specifications contain features (e.g., a colonial) that she does not particularly like. At this point, the user might leverage this example as an anchor and specify the particular attributes in it that she wants to be different. Note that the reason that the user is able to make this second set of critiqued query specifications is that she now has a concrete example to work with that she was not aware of earlier. The interfaces for critiquing can be defined in a number of different ways, and they are discussed in detail in section 5.3.2. The system then issues a new query with the modified target, and with a *reduced* set of candidates, which were the results from the previous query. In many cases, the effect is to simply prune the search results of cases that are not considered relevant, rather than provide a re-ranking of the returned results. Therefore, unlike constraint-based systems, the number of returned responses in case-based iterations generally reduces from one cycle to the next. However, it is also possible to design case-based systems in which the candidates are not always reduced from one iteration to the next by expanding the scope of each query to the entire database, rather than the currently retrieved set of candidate results. This type of design choice has its own trade-offs. For example, by expanding the scope of each query, the user will be able to navigate to a final result that is more distant from the current query. On the other hand, it is also possible that the results might become increasingly irrelevant in later iterations. For the purpose of this chapter, we assume that the returned candidates always reduce from one iteration to the next.

Through repeated critiquing, the user may sometimes arrive at a final result that is quite different from the initial query specification. After all, it is often difficult for a user to articulate *all* their desired features at the very beginning. For example, the user might not be aware of an acceptable price point for the desired home features at the beginning of the querying process. This interactive approach bridges the gap between her initial understanding and item availability. It is this power of assisted browsing that makes case-based methods so powerful in increasing user awareness. It is sometimes also possible for the user to arrive at an empty set of candidates through repeated reduction of the candidate set. Such a session may be viewed as a fruitless session, and in this case, the user has to restart from scratch at the entry point. Note that this is different from constraint-based systems, where a user also has the option of relaxing their current set of requirements to enlarge the result set. The reason for this difference is that case-based systems generally reduce the number of candidates from one cycle to the next, whereas constraint-based systems do not.

In order for a case-based recommender system to work effectively, there are two crucial aspects of the system that must be designed effectively:

1. *Similarity metrics:* The effective design of similarity metrics is very important in case-based systems in order to retrieve relevant results. The importance of various attributes must be properly incorporated within the similarity function for the system to work effectively.

2. *Critiquing methods:* The interactive exploration of the item space is supported with the use of critiquing methods. A variety of different critiquing methods are available to support different exploration goals.

In this section, we will discuss both these important aspects of case-based recommender system design.

5.3.1 Similarity Metrics

The proper design of similarity metrics is essential in retrieving meaningful items in response to a particular query. The earliest *FindMe* systems [121] ordered the attributes in decreasing level of importance and first sorted on the most important criterion, then the next most important, and so on. For example, in the *Entree* restaurant recommender system, the first sort might be based on the cuisine type, the second on the price, and so on. While this approach is efficient, its usage may not be effective for every domain. In general, it is desirable to develop a closed-form similarity function whose parameters can either be set by domain experts, or can be tweaked by a learning process.

Consider an application in which the product is described by d attributes. We would like to determine the similarity values between two *partial* attribute vectors defined on a subset S of the universe of d attributes (i.e., $|S| = s \leq d$). Let $\overline{X} = (x_1 \ldots x_d)$ and $\overline{T} = (t_1 \ldots t_d)$ represent two d-dimensional vectors, which might be partially specified. Here, $\overline{T}$ represents the target. It is assumed that at least the attribute subset $S \subseteq \{1 \ldots d\}$ is specified in both vectors. Note that we are using *partial* attribute vectors because such queries are often defined only on a small subset of attributes specified by the user. For example, in the aforementioned real estate example, the user might specify only a small set of query features, such as the number of bedrooms or bathrooms. Then, the similarity function $f(\overline{T}, \overline{X})$ between the two sets of vectors is defined as follows:

$$f(\overline{T}, \overline{X}) = \frac{\sum_{i \in S} w_i \cdot Sim(t_i, x_i)}{\sum_{i \in S} w_i} \tag{5.2}$$

Here, $Sim(t_i, x_i)$ represents the similarity between the values x_i and y_i. The weight w_i represents the weight of the ith attribute, and it regulates the relative importance of that attribute. How can the similarity functions $Sim(t_i, x_i)$ and the attribute importance w_i be learned?

Fist, we will discuss the determination of the similarity function $Sim(t_i, x_i)$. Note that these attributes might be either quantitative or categorical, which further adds to the heterogeneity and complexity of such a system. Furthermore, attributes might be symmetric or asymmetric in terms of higher or lower values [558]. For example, consider the price attribute in the home-buying example of Table 5.2. If a returned product has a lower price than the target value, then it is more easily acceptable than a case in which the returned product has a larger price than the target value. The precise level of asymmetry may be different for different attributes. For example, for an attribute, such as the camera resolution, the user might find larger resolutions more desirable, but the preference might not be quite as strong as in the case of the price. Other attributes might be completely symmetric, in which case the user would want the attribute value exactly at the target value t_i. An example of a symmetric metric is as follows:

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i} \tag{5.3}$$

Here, max_i and min_i represent the maximum or minimum possible values of the attribute i. Alternatively, one might use the standard deviation σ_i (on historical data) to set the similarity function:

$$Sim(t_i, x_i) = \max\left\{0, 1 - \frac{|t_i - x_i|}{3 \cdot \sigma_i}\right\} \tag{5.4}$$

Note that in the case of the symmetric metric, the similarity is entirely defined by the difference between the two attributes. In the case of an asymmetric attribute, one can add an additional *asymmetric* reward, which kicks in depending on whether the target attribute value is smaller or larger. For the case of attributes in which larger values are better, an example of a possible similarity function is as follows:

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i} + \underbrace{\alpha_i \cdot I(x_i > t_i) \cdot \frac{|t_i - x_i|}{max_i - min_i}}_{\text{Asymmetric reward}} \tag{5.5}$$

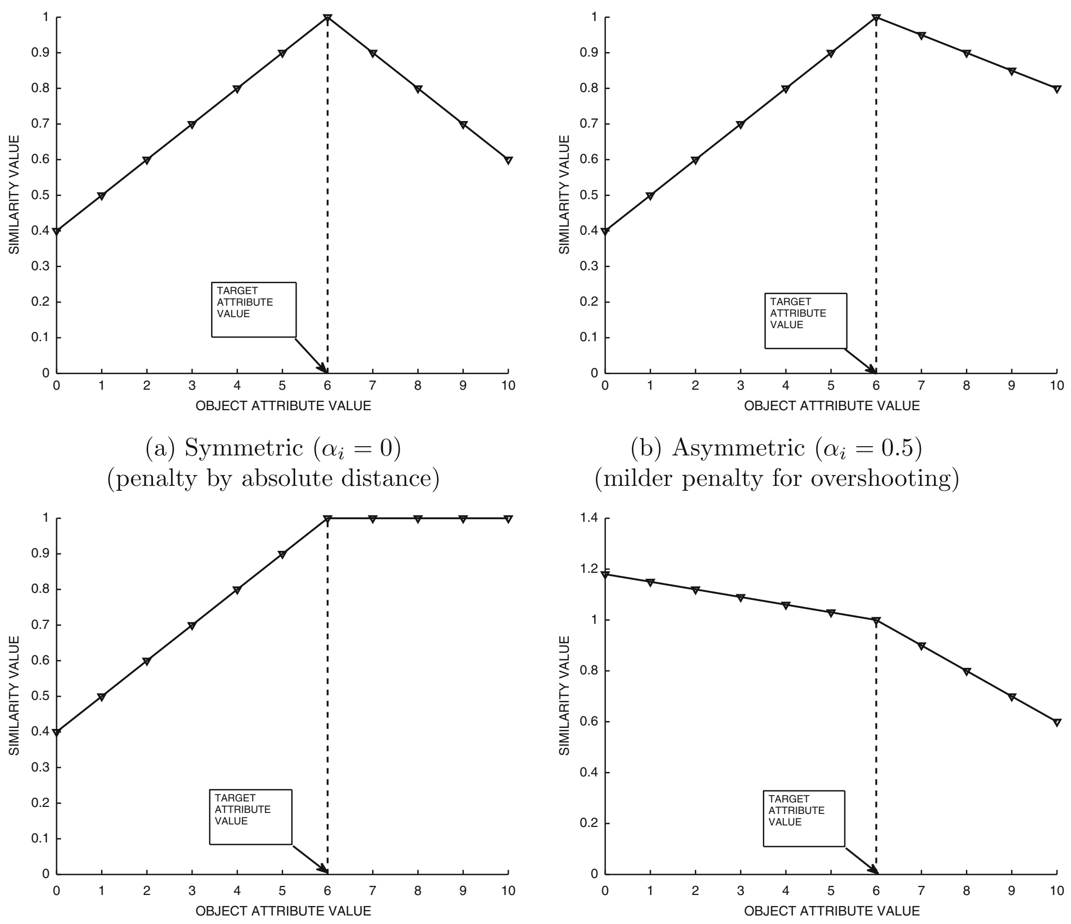
Here, $\alpha_i \geq 0$ is a user-defined parameter, and $I(x_i > t_i)$ is an indicator function that takes on the value of 1 if $x_i > t_i$, and 0 otherwise. Note that the reward kicks in only when the attribute value x_i (e.g., camera resolution) is greater than the target value t_i. For cases in which smaller values are better (e.g., price), the reward function is similar, except that smaller values are rewarded by the indicator function:

$$Sim(t_i, x_i) = 1 - \frac{|t_i - x_i|}{max_i - min_i} + \underbrace{\alpha_i \cdot I(x_i < t_i) \cdot \frac{|t_i - x_i|}{max_i - min_i}}_{\text{Asymmetric reward}} \tag{5.6}$$

The values of α_i are chosen in a highly domain-specific way. For values of $\alpha_i > 1$, the "similarity" actually increases with greater distance to the target. In such cases, it is helpful to think of $Sim(t_i, x_i)$ as a *utility* function rather than as a similarity function. For example, in the case of price, one would always prefer a lower price to a higher price, although the target price might define an inflection point in the strength with which one prefers a lower price to a higher price. When the value of α_i is exactly 1.0, it implies that one does not care about further change from the target value in one of the directions. An example might be the case of camera resolution, where one might not care about resolutions beyond a certain point. When $\alpha_i \in (0, 1)$, it implies that the user prefers a value at the target over all other values but she may have asymmetric preferences on either side of the target. For example, a user's preference for horsepower might strongly increase up to the target, and she might also have a mild aversion to a horsepower greater than the target because of greater fuel consumption. These examples suggest that there are no simple ways of pre-defining such similarity metrics; a lot of work needs to be done by the domain expert.

Examples of symmetric and asymmetric similarity functions are illustrated in Figure 5.6. The domain range is $[0, 10]$, and a target value of 6 is used. A symmetric similarity function is shown in Figure 5.6(a), where the similarity is linearly dependent on the distance from the target. However, in the horsepower example discussed above, the asymmetric similarity function of Figure 5.6(b) might be more appropriate, where $\alpha_i = 0.5$. For an attribute such as camera resolution, one might decide to allocate no utility beyond the user's target, as a result of which the similarity function might be flat beyond that point. Such a case is illustrated in Figure 5.6(c), where α_i is set to 1. Finally, in the case of price, smaller values are rewarded, although the user's target price might define an inflection point in the utility function. This case is illustrated in Figure 5.6(d), where the value of α_i is set to 1.3, with rewards being awarded for undershooting the target. This particular case is noteworthy, because the "similarity" is actually increasing with greater distance from the target as long as the value is as small as possible. In such cases, the utility interpretation of such functions makes a lot more sense than the similarity interpretation. In this interpretation, the target attribute values represent only key inflection points of the utility function.

For the case of categorical data, the determination of similarity values is often more challenging. Typically, domain hierarchies are constructed in order to determine the similarity values. Two objects that are closer to one another within the context of a domain hierarchy may be considered more similar. This domain hierarchy is sometimes directly available from sources such as the North American Industry Classification System (NAICS), and in other cases it needs to be directly constructed by hand. For example, an attribute such as the movie genre can be classified hierarchically, as shown in Figure 5.7. Note that related genres tend to be closer to one another in the hierarchy. For example, movies for children are considered to be so different from those for general audiences that they bifurcate at the root of the taxonomy. This hierarchy may be used by the domain expert to hand-code similarities. In some cases, learning methods can also be used to facilitate the similarity computation. For example, feedback could be elicited from users about pairs of genres, and learning methods could be used to learn the similarity between pairs of items [18]. The broader learning approach can also be used to determine other parameters of the similarity function, such as the value of α_i in Equations 5.5 and 5.6. It is noteworthy that the specific form of the similarity function may be different from that in Equations 5.5 and 5.6, depending on the data domain. It is here that the domain expert has to invest a significant amount of time in deciding how to model the specific problem setting. This investment is an inherent part of the domain-specific effort that knowledge-based recommender systems demand, and also derive their name from.



(a) Symmetric ($\alpha_i = 0$)
(penalty by absolute distance)

(b) Asymmetric ($\alpha_i = 0.5$)
(milder penalty for overshooting)

(a) Asymmetric ($\alpha_i = 1.0$)
(no penalty for overshooting)

(b) Asymmetric ($\alpha_i = 1.3$)
(less is always better)

Figure 5.6: Examples of different types of symmetric and asymmetric similarity

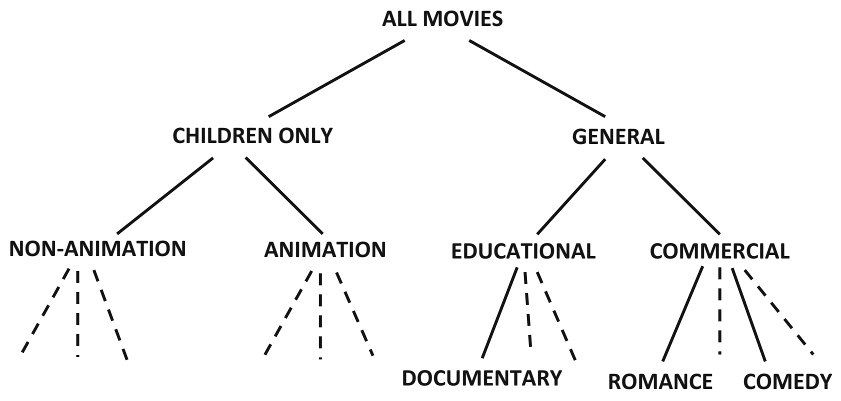


Figure 5.7: An example of hierarchical classification of movie genres

A second issue in the design of similarity functions is the determination of the relative importance of various attributes. The relative importance of the ith attributes is regulated by the parameter w_i in Equation 5.2. One possibility is for a domain expert to hand-code the values of w_i through trial and experience. The other possibility is to learn the values of w_i with user feedback. Pairs of target objects could be presented to users, and users might be asked to rate how similar these target objects are. This feedback can be used in conjunction with a linear regression model to determine the value of w_i. Linear regression models are discussed in detail in section 4.4.5 of Chapter 4, and their usage for similarity function learning is discussed in [18]. A number of other results [97, 163, 563, 627] discuss learning methods with user feedback in the specific context of recommender systems. Many of these methods, such as those in [627], show how feature weighting can be achieved with user feedback. The work in [563] elicits feedback from the user in terms of the relative *ordering* of the returned cases, and uses it to learn the relative feature weights. It is often easier for the user to specify relative orderings rather than to specify explicit similarity values for pairs of objects.

5.3.1.1 Incorporating Diversity in Similarity Computation

As case-based systems use item attributes to retrieve similar products, they face many of the same challenges as content-based systems in returning diverse results. In many cases, the results returned by case-based systems are all very similar. The problem with the lack of diversity is that if a user does not like the top-ranked result, she will often not like the other results, which are all very similar. For example, in the home buying application, it is possible for the recommendation system to return condominium units from the same complex under the same management. Clearly, this scenario reduces the true *choice* available to the user among the top ranked results.

Consider a scenario where it is desired to retrieve the top-k results matching a particular case. One possibility is to retrieve the top $b \cdot k$ results (for $b > 1$) and then randomly select k items from this list. This strategy is also referred to as the *bounded random selection strategy*. However, such a strategy does not seem to work very well in practice.

A more effective approach is the *bounded greedy selection strategy* [560]. In this strategy, we start with the top $b \cdot k$ cases similar to the target, and incrementally build a diverse set of k instances from these $b \cdot k$ cases. Therefore, we start with the empty set R and incrementally build it by adding instances from the base set of $b \cdot k$ cases. The first step is to create a *quality* metric that combines similarity and diversity. Assume without loss of generality that the similarity function $f(\overline{X}, \overline{Y})$ always maps to a value in $(0, 1)$. Then, the diversity $D(\overline{X}, \overline{Y})$ can be viewed as the distance between $\overline{X}$ and $\overline{Y}$:

$$D(\overline{X}, \overline{Y}) = 1 - f(\overline{X}, \overline{Y}) \tag{5.7}$$

Then, the average diversity between the candidate $\overline{X}$, and a set R of currently selected cases is defined as the average diversity between $\overline{X}$ and cases in R:

$$D^{avg}(\overline{X}, R) = \frac{\sum_{\overline{Y} \in R} D(\overline{X}, \overline{Y})}{|R|} \tag{5.8}$$

Then, for target $\overline{T}$, the overall quality $Q(\overline{T}, \overline{X}, R)$ is computed as follows:

$$Q(\overline{T}, \overline{X}, R) = f(\overline{T}, \overline{X}) \cdot D^{avg}(\overline{X}, R) \tag{5.9}$$

The case $\overline{X}$ with the greatest quality is incrementally added to the set R until the cardinality of the set R is k. This set is presented to the user. Refer to the bibliographic notes for other diversity enhancing techniques used in the literature.

5.3.2 Critiquing Methods

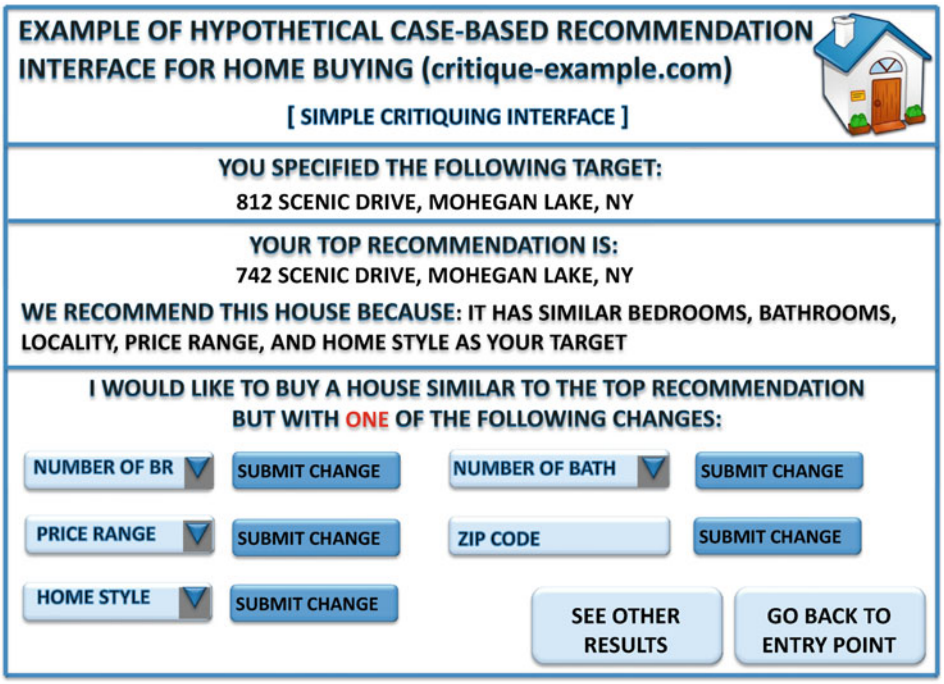
Critiques are motivated by the fact that users are often not in a position to state their requirements exactly in the initial query. In some complex domains, they might even find it difficult to translate their needs in a semantically meaningful way to the attribute values in the product domain. It is only after viewing the results of a query that a user might realize that she should have couched her query somewhat differently. Critiques are designed to provide the users this ability after the fact.

After the results have been presented to the users, feedback is typically enabled through the use of *critiques*. In many cases, the interfaces are designed to critique the most similar matching item, although it is technically possible for the user to critique any of the items on the retrieved list of k items. In critiques, the users specify *change* requests on one or more attributes of an item that they may like. For example, in the home-buying application of Figure 5.2, the user might like a particular house, but she may want the house in a different locality or with one more bedroom. Therefore, the user may specify the changes in the features of one of the items she likes. The user may specify a *directional critique* (e.g., "cheaper") or a *replacement critique* (e.g., "different color"). In such cases, examples that do not satisfy the user-specified critiques are eliminated, and examples similar to the user-preferred item (but satisfying the current sequence of critiques) are retrieved. When multiple critiques are specified in sequential recommendation cycles, preference is given to more recent critiques.

At a given moment in time, the user may specify either a single feature or a combination of features for modification. In this context, the critiques are of three different types, corresponding to *simple critiques*, *compound critiques*, and *dynamic critiques*. We will discuss each of these types of critiques in the following sections.

5.3.2.1 Simple Critiques

In a simple critique, the user specifies a single change to one of the features of a recommended item. In Figure 5.8, we have used our earlier case-based scenario (`critique-example.com`) to show an example of a simple critiquing interface. Note that the user can specify a change to only one of the features of the recommended house in this interface. Often, in many systems, such as *FindMe* systems, a more conversational interface is used, where users specify whether to increase or decrease a specific attribute value rather than explicitly modify one of the target attribute values. This is referred to as a *directional critique*. In such cases, the candidate list is simply pruned of those objects for which the critiqued attribute is on the wrong side of the user's stated preference. The advantage of such an approach is that the user is able to state her preference and navigate through the product space without having to specify or change attribute values in a precise way. Such an approach is particularly important in domains where the users might not know the exact value of the attribute to use (e.g., the horsepower of an engine). Another advantage of a directional critique is that it has a simple conversational style, which might be more intuitive and appealing to the user. In cases where the user does not find the current set of retrieved results to be useful at all, she also has the option of going back to the entry point. This represents a fruitless cycle through the critiquing process.


EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)

[SIMPLE CRITIQUING INTERFACE]

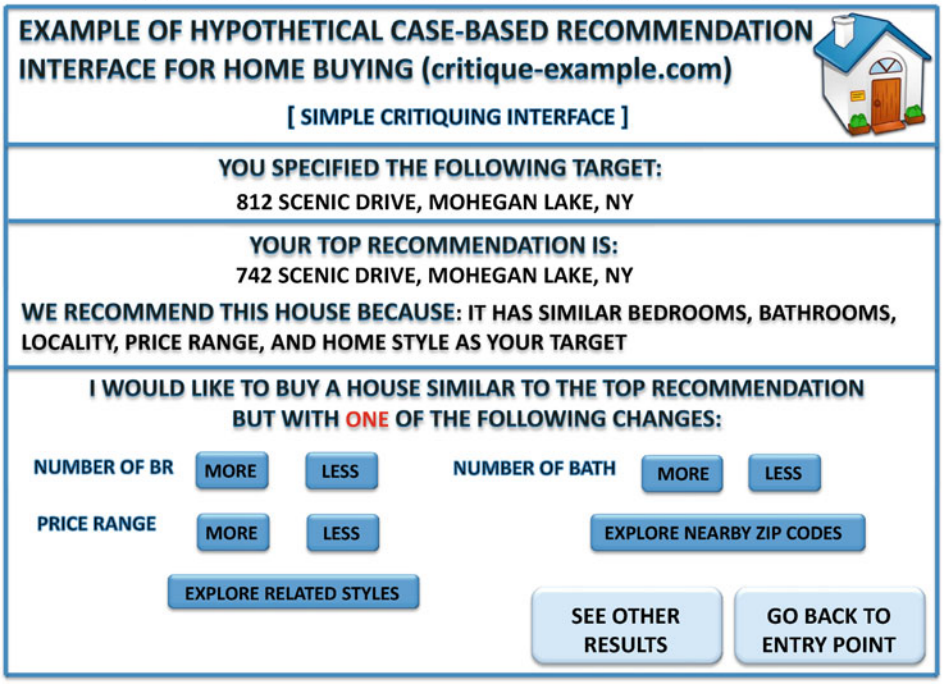
YOU SPECIFIED THE FOLLOWING TARGET:
812 SCENIC DRIVE, MOHEGAN LAKE, NY

YOUR TOP RECOMMENDATION IS:
742 SCENIC DRIVE, MOHEGAN LAKE, NY

WE RECOMMEND THIS HOUSE BECAUSE: IT HAS SIMILAR BEDROOMS, BATHROOMS, LOCALITY, PRICE RANGE, AND HOME STYLE AS YOUR TARGET

I WOULD LIKE TO BUY A HOUSE SIMILAR TO THE TOP RECOMMENDATION BUT WITH ONE OF THE FOLLOWING CHANGES:

(a) Simple critiquing by directly modifying feature values


EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)

[SIMPLE CRITIQUING INTERFACE]

YOU SPECIFIED THE FOLLOWING TARGET:
812 SCENIC DRIVE, MOHEGAN LAKE, NY

YOUR TOP RECOMMENDATION IS:
742 SCENIC DRIVE, MOHEGAN LAKE, NY

WE RECOMMEND THIS HOUSE BECAUSE: IT HAS SIMILAR BEDROOMS, BATHROOMS, LOCALITY, PRICE RANGE, AND HOME STYLE AS YOUR TARGET

I WOULD LIKE TO BUY A HOUSE SIMILAR TO THE TOP RECOMMENDATION BUT WITH ONE OF THE FOLLOWING CHANGES:

(b) The conversational style of directional critiques

Figure 5.8: Hypothetical examples of user interfaces for simple critiquing in a case-based recommender (`critique-example.com`)

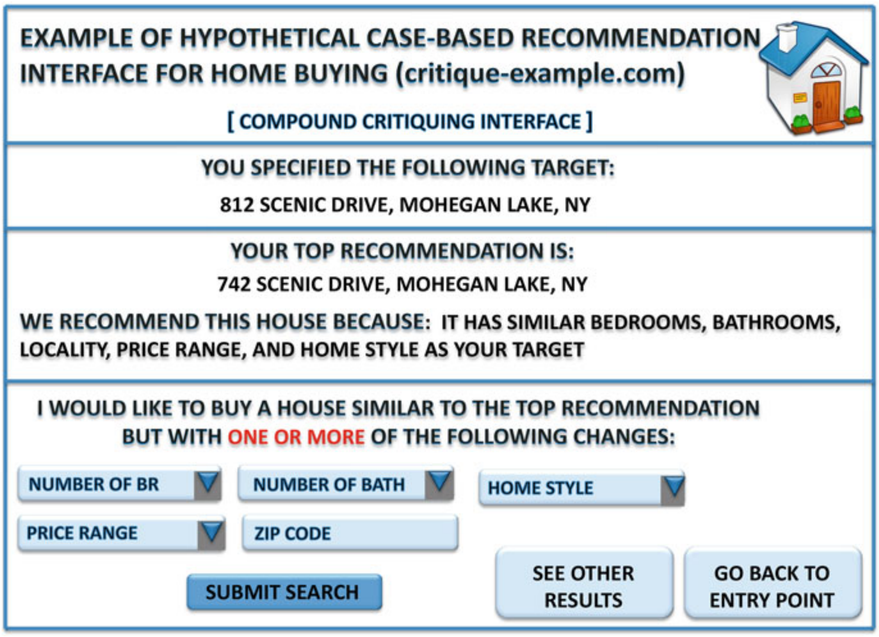
The main problem with the simple critiquing approach is its laborious navigation. If the recommended product contains many features that are required to be changed, then it will lead to a longer chain of subsequent critiques. Furthermore, when one of the features is changed, the recommender system may automatically need to change at least some of the other feature values depending on item availability. In most cases, it is impossible to hold the other feature values at exactly constant values in a given cycle. As a result, when the user has changed a few features to their desired values, they may realize that the other feature values are no longer acceptable. The larger the number of recommendation cycles, the less the control that the user will have on changes in the other feature values that were acceptable in earlier iterations. This problem often results from the user's lack of understanding about the natural trade-offs in the problem domain. For example, a user might not understand the trade-off between horsepower and fuel efficiency and attempt to navigate to a car with high horsepower and also a high fuel efficiency of 50 miles to the gallon [121]. This problem of fruitlessness in long recommendation cycles is discussed in detail in [423]. The main problem in many critiquing interfaces is that the next set of recommended items are based on the *most recent* items being critiqued, and there is no way of navigating back to earlier items. As a result, a long cycle of simple critiques may sometimes drift to a fruitless conclusion.

5.3.2.2 Compound Critiques

Compound critiques were developed to reduce the length of recommendation cycles [414]. In this case, the user is able to specify multiple feature modifications in a single cycle. For example, the *Car Navigator* system [120] allows the user to specify multiple modifications, which are hidden behind informal descriptions that the user can understand (e.g., *classier, roomier, cheaper, sportier*). For example, the domain expert might encode the fact that "*classier*" suggests a certain subset of models with increased price and sophisticated interior structure. Of course, it is also possible for the user to modify the required product features directly, but it increases the burden on her. The point in conversational critiquing is that when a user might wish to have a "*classier*" car, but they might not be easily able to concretely express it in terms of the product features such as the interior structure of the car. On the other hand, a qualification such as "*classier*" is more intuitive, and it can be encoded in terms of the product features by a domain expert. This interactive process is designed to help them learn the complex product space in an intuitive way.

In the home-buying example of Table 5.2, the user might specify a different locality and change in the price in a single cycle. An example of a compound critiquing example for the home-buying example is illustrated in Figure 5.9(a). To make the approach more conversational, an interface like the one in Figure 5.9(b), will automatically encode multiple changes within a single selection. For example, if the user selects "*roomier*," it implies that both the number of bedrooms and the number of bathrooms might need to be increased. For the second type of interface, the domain expert has to expend significant effort in designing the relevant interface and the interpretation of user choices in terms of changes made to multiple product features. This encoding is static, and it is done up front.

The main advantage of compound critiquing is that the user can change multiple features in the target recommendation in order to issue a new query or prune the search results from the previous query. As a result, this approach allows large jumps through the product feature space, and the user often has better control over the critiquing process. This is useful for reducing the number of recommendation cycles and making the exploration process more efficient. It is, however, not clear whether compound critiques always help a user learn the



EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)

[COMPOUND CRITIQUING INTERFACE]

YOU SPECIFIED THE FOLLOWING TARGET:

812 SCENIC DRIVE, MOHEGAN LAKE, NY

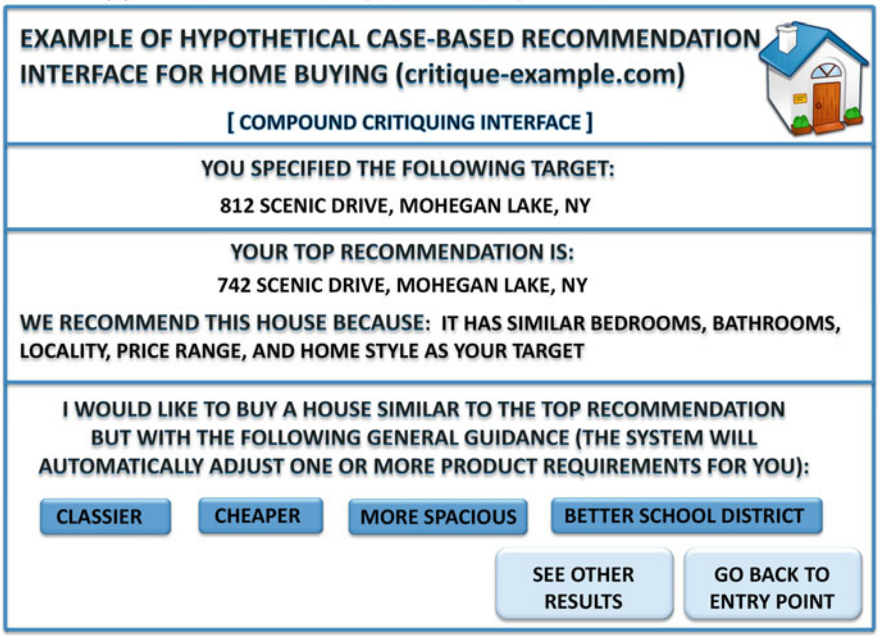
YOUR TOP RECOMMENDATION IS:

742 SCENIC DRIVE, MOHEGAN LAKE, NY

WE RECOMMEND THIS HOUSE BECAUSE: IT HAS SIMILAR BEDROOMS, BATHROOMS, LOCALITY, PRICE RANGE, AND HOME STYLE AS YOUR TARGET

I WOULD LIKE TO BUY A HOUSE SIMILAR TO THE TOP RECOMMENDATION BUT WITH ONE OR MORE OF THE FOLLOWING CHANGES:

(a) Compound critiquing by modifying multiple feature values



EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)

[COMPOUND CRITIQUING INTERFACE]

YOU SPECIFIED THE FOLLOWING TARGET:

812 SCENIC DRIVE, MOHEGAN LAKE, NY

YOUR TOP RECOMMENDATION IS:

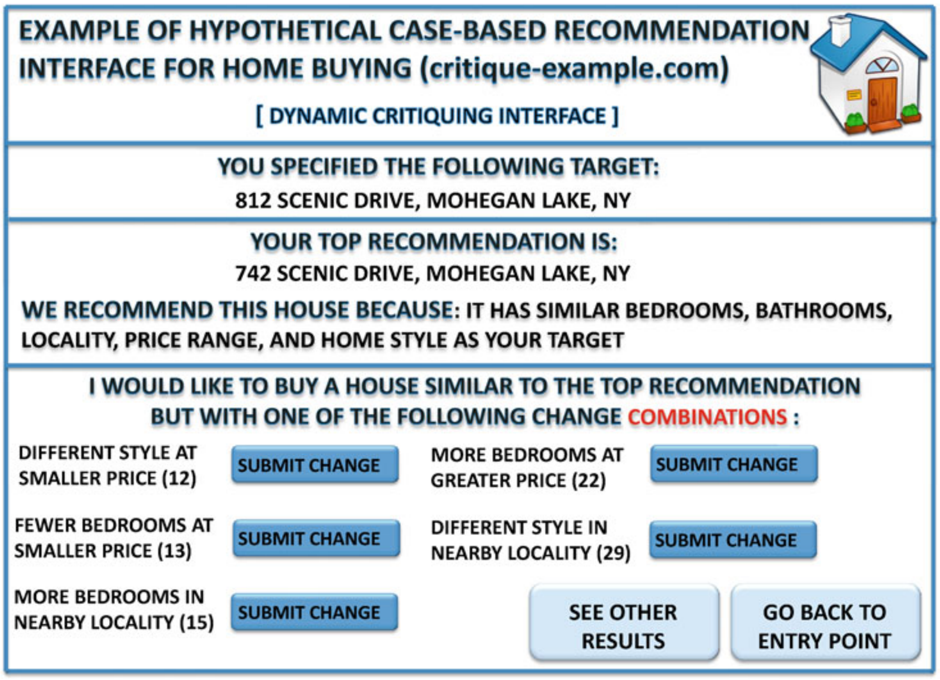
742 SCENIC DRIVE, MOHEGAN LAKE, NY

WE RECOMMEND THIS HOUSE BECAUSE: IT HAS SIMILAR BEDROOMS, BATHROOMS, LOCALITY, PRICE RANGE, AND HOME STYLE AS YOUR TARGET

I WOULD LIKE TO BUY A HOUSE SIMILAR TO THE TOP RECOMMENDATION BUT WITH THE FOLLOWING GENERAL GUIDANCE (THE SYSTEM WILL AUTOMATICALLY ADJUST ONE OR MORE PRODUCT REQUIREMENTS FOR YOU):

(b) Reducing the user's burden of specifying multiple features with domain knowledge

Figure 5.9: Hypothetical examples of user interfaces for compound critiquing in a case-based recommender (`critique-example.com`)


EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)

[DYNAMIC CRITIQUING INTERFACE]

YOU SPECIFIED THE FOLLOWING TARGET:
812 SCENIC DRIVE, MOHEGAN LAKE, NY

YOUR TOP RECOMMENDATION IS:
742 SCENIC DRIVE, MOHEGAN LAKE, NY

WE RECOMMEND THIS HOUSE BECAUSE: IT HAS SIMILAR BEDROOMS, BATHROOMS, LOCALITY, PRICE RANGE, AND HOME STYLE AS YOUR TARGET

I WOULD LIKE TO BUY A HOUSE SIMILAR TO THE TOP RECOMMENDATION BUT WITH ONE OF THE FOLLOWING CHANGE COMBINATIONS :

DIFFERENT STYLE AT SMALLER PRICE (12) SUBMIT CHANGE
MORE BEDROOMS AT GREATER PRICE (22) SUBMIT CHANGE
FEWER BEDROOMS AT SMALLER PRICE (13) SUBMIT CHANGE
DIFFERENT STYLE IN NEARBY LOCALITY (29) SUBMIT CHANGE
MORE BEDROOMS IN NEARBY LOCALITY (15) SUBMIT CHANGE

SEE OTHER RESULTS
GO BACK TO ENTRY POINT

Figure 5.10: A hypothetical example of a user interface for dynamic critiquing in a case-based recommender (`critique-example.com`)

product space better than simple critiques; short critiquing cycles also reduce the likelihood of the user learning different trade-offs and correlations between features in the product space. On the other hand, a user may sometimes learn a lot about the product space by going through the slow and laborious process of simple critiquing.

5.3.2.3 Dynamic Critiques

Although compound critiques allow larger jumps through the navigation space, they do have the drawback that the critiquing options presented to the user are *static* in the sense that they do not depend on the retrieved results. For example, if the user is browsing cars, and she is already browsing the most expensive car with the largest horsepower possible, the option to increase the horsepower and the price will still be shown in the critiquing interface. Clearly, specifying these options will lead to a fruitless search. This is because users are often not fully aware of the inherent trade-offs in the complex product space.

In dynamic critiquing, the goal is to use data mining on the retrieved results to determine the most fruitful avenues of exploration and present them to the user. Thus, dynamic critiques are, by definition, compound critiques because they almost always represent combinations of changes presented to the user. The main difference is that only the subset of the most relevant *possibilities* are presented, based on the currently retrieved results. Therefore, dynamic critiques are designed to provide better guidance to the user during the search process.

An important aspect of dynamic critiquing is the ability to discover frequent combinations of product feature changes. The notion of *support* is adapted from frequent pattern mining [23] in order to determine patterns of frequently co-occurring product features in the retrieved results. The support of a pattern is defined as the fraction of the retrieved results that satisfy that pattern. Refer to Definition 3.3.1 in Chapter 3 for a formal definition of support. Therefore, this approach determines all the patterns of change that specify a predefined minimum support value. For example, in the home-buying example of Table 5.2, the system might determine the following dynamic critiques in order of support:

More Bedrooms, Greater Price: Support= 25%
More Bedrooms, More Bathrooms, Greater Price: Support= 20%
Fewer Bedrooms, Smaller Price: Support= 20%
More Bedrooms, Locality=Yonkers: Support= 15%

Note that conflicting options such as "*More Bedrooms, Smaller Price*" have a smaller chance of being included because they might be eliminated based on the minimum support criterion. However, low support patterns are not necessarily uninteresting. In fact, once all the patterns satisfying the minimum support threshold have been determined, many recommender systems order the critiques to the user in ascending order of support. The logic for this approach is that low support critiques are often less obvious patterns that can be used to eliminate a larger number of items from the candidate list. A hypothetical example of a dynamic critiquing interface, based on our earlier home-buying system (`critique-example.com`), is illustrated in Figure 5.10. Note that a numerical quantity is associated with each of the presented options in the interface. This number corresponds to the raw support of the presented options.

A real-world example of a dynamic critiquing approach that uses frequent pattern and association rule mining is the *Qwikshop* system discussed in [491]. An important observation about dynamic critiquing systems is that they increase the cognitive load on the user, when viewed on a *per-cycle basis*, but they reduce the cognitive load over the course of the *entire session* because of their ability to arrive at acceptable recommendations more quickly [416]. This is one of the reasons that the effective design of explanatory processes into the critiquing cycle is more important in dynamic critiquing systems.

5.3.3 Explanation in Critiques

It is always advisable to build explanatory power into the critiquing process, because it helps the user understand the information space better. There are several forms of explanation that are used to improve the quality of critiques. Some examples of such explanations are as follows:

1. In simple critiquing, it is common for a user to navigate in a fruitless way because of a lack of awareness of the inherent trade-offs in the product space. For example, a user might successively increase the horsepower, increase the mileage per gallon, and then try to reduce the desired price. In such cases, the system might not be able to show an acceptable result to the user, and the user will have to start the navigation process afresh. At the end of such a session, it is desirable for the system to automatically determine the nature of the trade-off that resulted in a fruitless session. It is often possible to determine such trade-offs with the use of correlation and co-occurrence statistics. The user can then be provided insights about the conflicts in the critiques entered by them in the previous session. Such an approach is used in some of the *FindMe* systems [121].

2. It has been shown in [492] how explanations can be used in conjunction with dynamic compound critiques during a session. For example, the *Qwikshop* system provides information about the fraction of the instances satisfying each compound critique. This provides the user with a clear idea of the size of the space they are about to explore *before* making a critiquing choice. Providing the user with better explanations *during* the session increases the likelihood that the session will be fruitful.

The main danger in critiquing-based systems is the likelihood of users meandering through the knowledge space in an aimless way without successfully finding what they are looking for. Adding explanations to the interface greatly reduces this likelihood.

5.4 Persistent Personalization in Knowledge-Based Systems

Although knowledge-based systems, such as constraint-based systems, allow the specification of user preferences, characteristics, and/or demographic attributes, the entered information is typically session-specific, and it is not *persistent* across sessions. The only persistent data in most such systems is the domain knowledge in the form of various system-specific databases, such as constraints or similarity metrics. This lack of persistent data is a natural consequence of how knowledge-based systems tend to use historical data only in a limited way compared to content-based and collaborative systems. This is also an advantage of knowledge-based systems, because they tend to suffer less from cold-start issues compared to other systems that are dependent on historical data. In fact, knowledge-based recommender systems are often designed for more expensive and occasionally bought items, which are highly customized. In such cases, historical data should be used with some caution, even when they are available. Nevertheless, a few knowledge-based systems have also been designed to use persistent forms of personalization.

The user's actions over various sessions can be used to build a persistent profile about the user regarding what they have liked or disliked. For example, *CASPER* is an online recruitment system [95] in which the user's actions on retrieved job postings, such as saving the advertisement, e-mailing it to themselves, or applying to the posting, are saved for future reference. Furthermore, users are allowed to negatively rate advertisements when they are irrelevant. Note that this process results in the building of an implicit feedback profile. The recommendation process is a two-step approach. In the first step, the results are retrieved based on the user requirements, as in the case of any knowledge-based recommender. Subsequently, the results are ranked based on similarity to previous profiles that the user has liked. It is also possible to include collaborative information by identifying other users with similar profiles, and using their session information in the learning process.

Many steps in knowledge-based systems can be personalized when user interaction data is available. These steps are as follows:

1. The learning of utility/similarity functions over various attributes can be personalized for both constraint-based recommenders (ranking phase) and in case-based recommenders (retrieval phase). When past feedback from a particular user is available, it is possible to learn the relative importance of various attributes for that user in the utility function.

2. The process of constraint suggestion (cf. section 5.2.5) for a user can be personalized if a significant number of sessions of that user are available.

3. Dynamic critiques for a user can be personalized if sufficient data are available from that user to determine relevant patterns. The only difference from the most common form of dynamic critiquing is that user-specific data are leveraged rather than all the data for determining the frequent patterns. It is also possible to include the sessions of users with similar sessions in the mining process to increase the collaborative power of the recommender.

Although there are many avenues through which personalization can be incorporated within the framework of knowledge-based recommendation, the biggest challenge is usually the unavailability of sufficient session data for a particular user. Knowledge-based systems are inherently designed for highly customized items in a complex domain space. This is the reason that the level of personalization is generally limited in knowledge-based domains.

5.5 Summary

Knowledge-based recommender systems are generally designed for domains in which the items are highly customized, and it is difficult for rating information to directly reflect greater preferences. In such cases, it is desirable to give the user greater control in the recommendation process through requirement specification and interactivity. Knowledge-based recommender systems can be either constraint-based systems, or they can be case-based systems. In constraint-based systems, users specify their requirements, which are combined with domain-specific rules to provide recommendations. Users can add constraints or relax constraints depending on the size of the results. In case-based systems, the users work with targets and candidate lists that are iteratively modified through the process of critiquing. For retrieval, domain-dependent similarity functions are used, which can also be learned. The modifications to the queries are achieved through the use of critiquing. Critiques can be simple, compound, or dynamic. Knowledge-based systems are largely based on user requirements, and they incorporate only a limited amount of historical data. Therefore, they are usually effective at handling cold-start issues. The drawback of this approach is that historical information is not used for "filling in the gaps." In recent years, methods have also been designed for incorporating a greater amount of personalization with the use of historical information from user sessions.

5.6 Bibliographic Notes

Surveys on various knowledge-based recommender systems and preference elicitation methods may be found in [197, 417]. Case-based recommender systems are reviewed in [102, 116, 377, 558]. Surveys of preference elicitation methods and critiquing may be found in [148, 149]. Constraint-based recommender systems are discussed in [196, 197]. Historically, constraint-based recommendation systems were proposed much later than case-based recommenders. In fact, the original paper by Burke [116] on knowledge-based recommender systems mostly describes case-based recommenders. However, some aspects of constraint-based recommenders are also described in this work. Methods for learning utility functions in the context of constraint-based recommender systems are discussed in [155, 531]. Methods for handing empty results in constraint-based systems, such as fast discovery of small conflicting sets, and minimal relaxations are discussed in [198, 199, 273, 274, 289, 419, 574]. These works also discuss how these conflicting sets may be used to provide explanations and repair diagnoses of the user queries. Popularity-based methods for selecting the next constraint attribute are discussed in [196, 389]. The selection of default values for the attribute constraints is discussed in [483]. A well-known constraint-based recommender system is the *VITA* recommender [201], which was built on the basis of the *CWAdvisor* system [200].

The problem of similarity function learning for case-based recommenders is discussed in [18, 97, 163, 563, 627]. The work in [563] is notable in that learns weights of various features for similarity computation. Reinforcement learning methods for learning similarity functions for case-based systems are discussed in [288, 506]. The bounded random selection and bounded greedy selection strategies for increasing the diversity of case-based recommender systems are discussed in [560]. The work in [550] also combines similarity with diversity like the bounded greedy approach, but it applies only diversity on the retrieved set of $b \cdot k$ cases, rather than creating a quality metric combining similarity and diversity. The notions of *similarity layers* and *similarity intervals* for diversity enhancement are discussed in [420]. A compromise-driven approach for diversity enhancement is discussed in [421]. The power of order-based retrieval for similarity diversification is discussed in [101]. Experimental results [94, 560] show the advantages of incorporating diversity into recommender systems. The issue of critiquing in case-based recommender systems is discussed in detail in [417, 422, 423]. Compound critiques were first discussed in [120], although the term was first coined in [414]. A comparative study of various compound critiquing techniques may be found in [664]. The use of explanations in compound critiques is discussed in [492].

The earliest case-based recommenders were proposed in [120, 121] in the context of the *Entree* restaurant recommender. The earliest forms of these systems were also referred to as *FindMe* systems [121], which were shown to be applicable to a wide variety of domains. The Wasabi personal shopper is a case-based recommender system and is discussed in [125]. Case-based systems have been used for travel advisory services [507], online recruitment systems [95], car sales (*Car Navigator*) [120], video sales (*Video Navigator*) [121], movies (*Pick A Flick*) [121], digital camera recommendations (e.g., *Qwikshop*) [279, 491], and rental property accommodation [263].

Most knowledge-based systems leverage user requirements and preferences, as specified in a single session. Therefore, if a different user enters the same input, they will obtain exactly the same result. Although such an approach provides better control to the user, and also does not suffer from cold-start issues, it tends to ignore historical data when they are available. Recent years have also witnessed an increase in long-term and persistent information about the user in knowledge-based recommender systems [95, 454, 558]. An example of such a system is the *CASPER* online recruitment system [95], which builds persistent user profiles for future recommendation. A personalized travel recommendation system with the use of user profiles is discussed in [170]. The sessions of similar users are leveraged for personalized travel recommendations in [507]. Such an approach not only leverages the target user's behavior but also the collaborative information available in a community of users. The work in [641] uses the critiquing information over multiple sessions in a collaborative way to build user profiles. Another relevant work is the *MAUT* approach [665], which is based on multi-attribute utility theory. This approach learns a utility preference function for each user based on their critiques in the previous sessions. Another example of persistent data that can be effectively used in such systems is demographic information. Although demographic recommender systems vary widely in their usage [117, 320], some of the demographic systems can also be considered knowledge-based systems, when profile association rules are used to interactively suggest preferences to users in an online fashion [31, 32]. Such systems allow progressive refinement of the queries in order to derive the most appropriate set of rules for a particular demographic group. Similarly, various types of utility-based recommendation and ranking techniques are used within the context of knowledge-based systems [74].

5.7 Exercises

1. Implement an algorithm to determine whether a set of customer-specified requirements and a set of rules in a knowledge base will retrieve an empty set from a product catalog. Assume that the antecedent and the consequent of each rule both contain a single constraint on the product features. Constraints on numerical attributes are in the form of inequalities (e.g., $\mathit{Price} \leq 30$), whereas constraints on categorical attributes are in the form of unit instantiations (e.g., *Color*=*Blue*). Furthermore, customer requirements are also expressed as similar constraints in the feature space.

2. Suppose you had data containing information about the utility values of a particular customer for a large set of items in a particular domain (e.g., cars). Assume that the utility value of the jth product is u_j ($j \in \{1 \dots n\}$). The items are described by a set of d numerical features. Discuss how you will use these data to rank other items in the same product domain for this customer.