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:

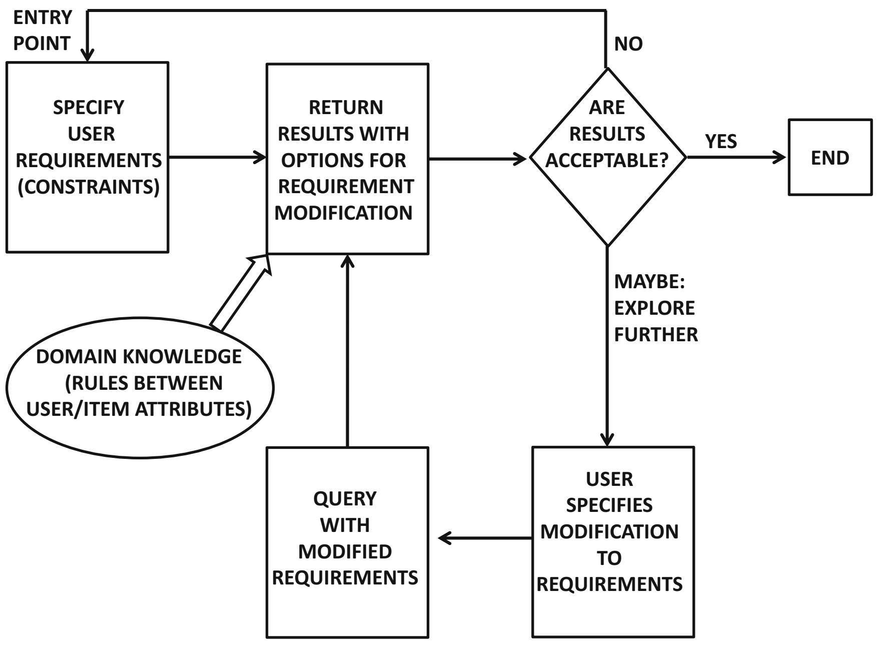
1. *Conversational systems:* In this case, the user preferences are determined in the context of a feedback loop. The main reason for this is that the item domain is complex, and the user preferences can be determined only in the context of an iterative conversational system.

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?"

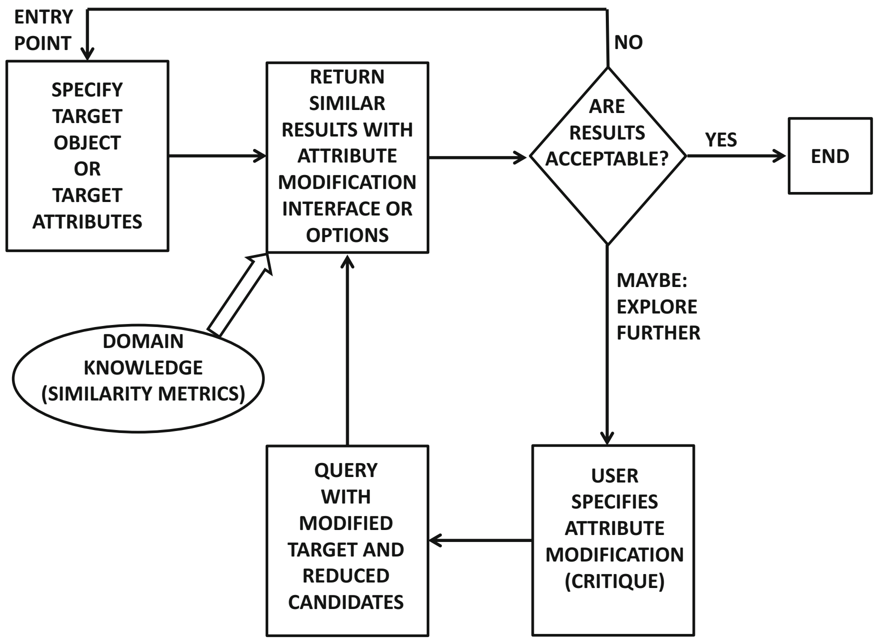
3. *Navigation-based recommendation:* In navigation-based recommendation, the user specifies a number of change requests to the item being currently recommended. Through an iterative set of change requests, it is possible to arrive at a desirable item. An example of a change request specified by the 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:

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

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:

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

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:

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

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:

     Suburban-or-rural=Suburban ⇒ *Locality=* ⟨*List of relevant localities* ⟩

     Min-Bedrooms≥3 ⇒ *Price≥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:

     Family-Size≥5 ⇒ *Min-Bedrooms≥3*

     Family-Size≥5 ⇒ *Min-Bathrooms≥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,

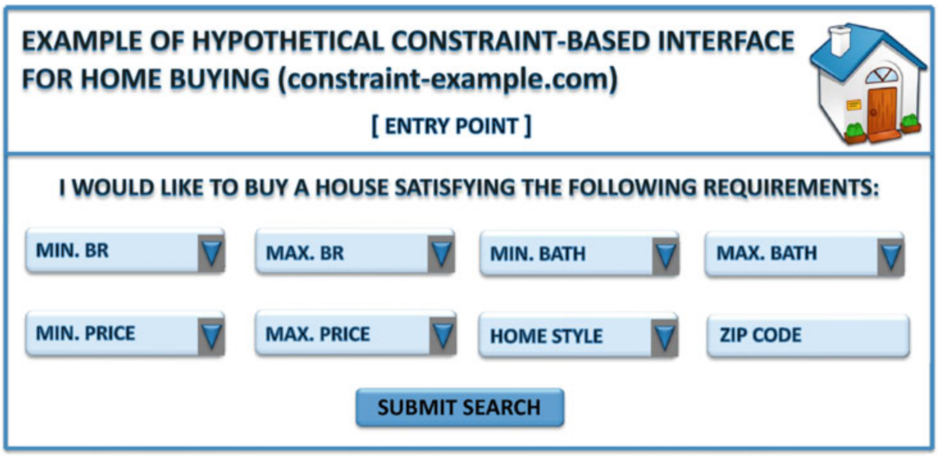
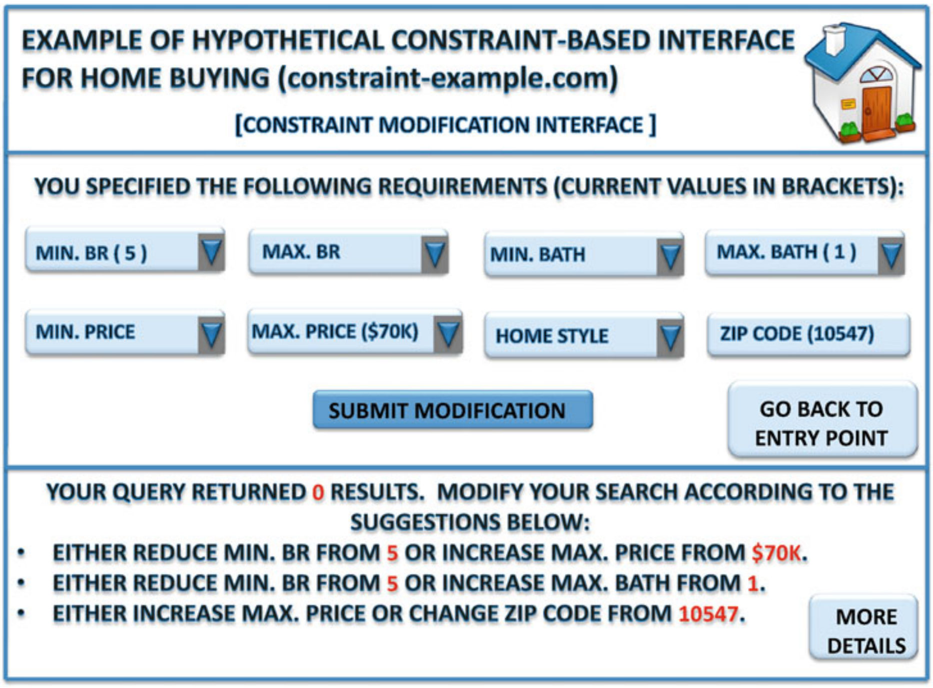

EXAMPLE OF HYPOTHETICAL CONSTRAINT-BASED INTERFACE FOR HOME BUYING (constraint-example.com)
[ENTRY POINT]
I WOULD LIKE TO BUY A HOUSE SATISFYING THE FOLLOWING REQUIREMENTS:
MIN. BR
MAX. BR
MIN. BATH
MAX. BATH
MIN. PRICE
MAX. PRICE
HOME STYLE
ZIP CODE
SUBMIT SEARCH

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):

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`)

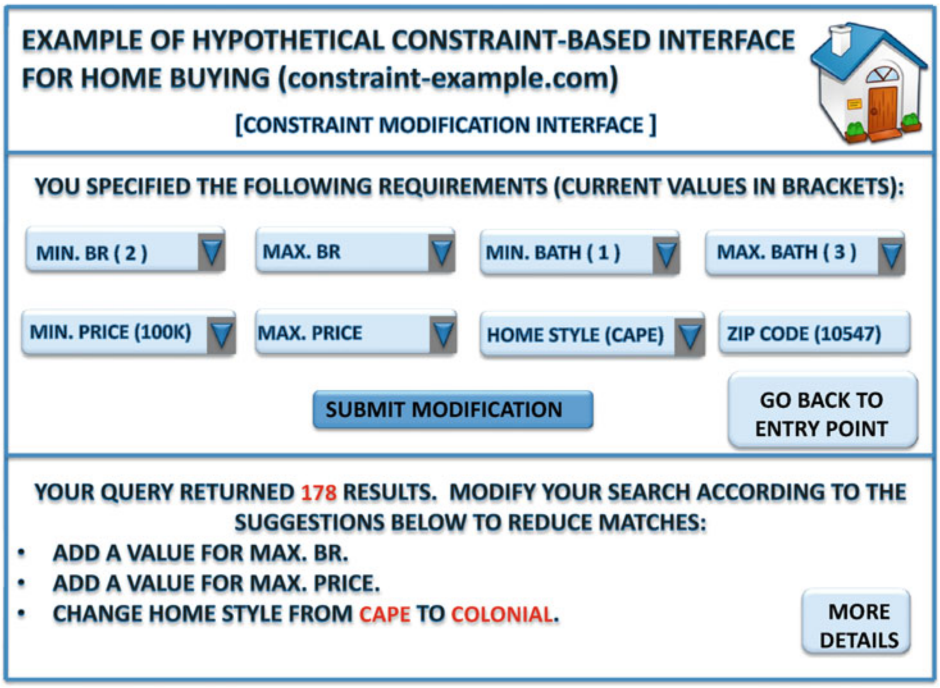
5.2.3 Ranking the Matched Items

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 $\overline{V} = (v_1 \dots v_d)$ 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):

MIN. BR (2) | MAX. BR | MIN. BATH (1) | MAX. BATH (3)

MIN. PRICE (100K) | MAX. PRICE | HOME STYLE (CAPE) | ZIP CODE (10547)

SUBMIT MODIFICATION | GO BACK TO ENTRY POINT

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.

MORE DETAILS

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

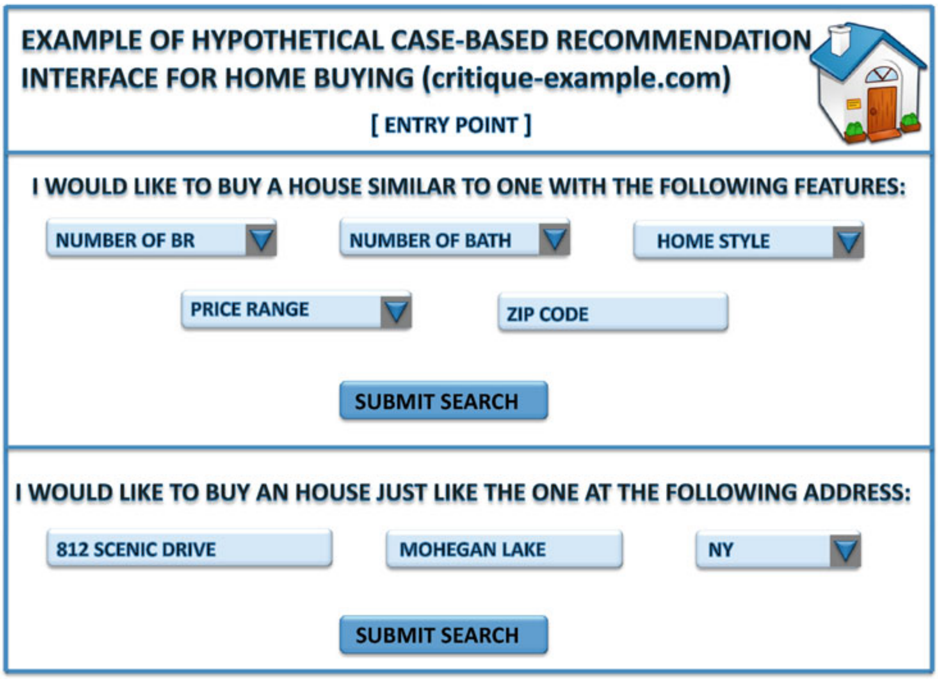

EXAMPLE OF HYPOTHETICAL CASE-BASED RECOMMENDATION INTERFACE FOR HOME BUYING (critique-example.com)
[ENTRY POINT]
I WOULD LIKE TO BUY A HOUSE SIMILAR TO ONE WITH THE FOLLOWING FEATURES:
NUMBER OF BR
NUMBER OF BATH
HOME STYLE
PRICE RANGE
ZIP CODE
SUBMIT SEARCH
I WOULD LIKE TO BUY AN HOUSE JUST LIKE THE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

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

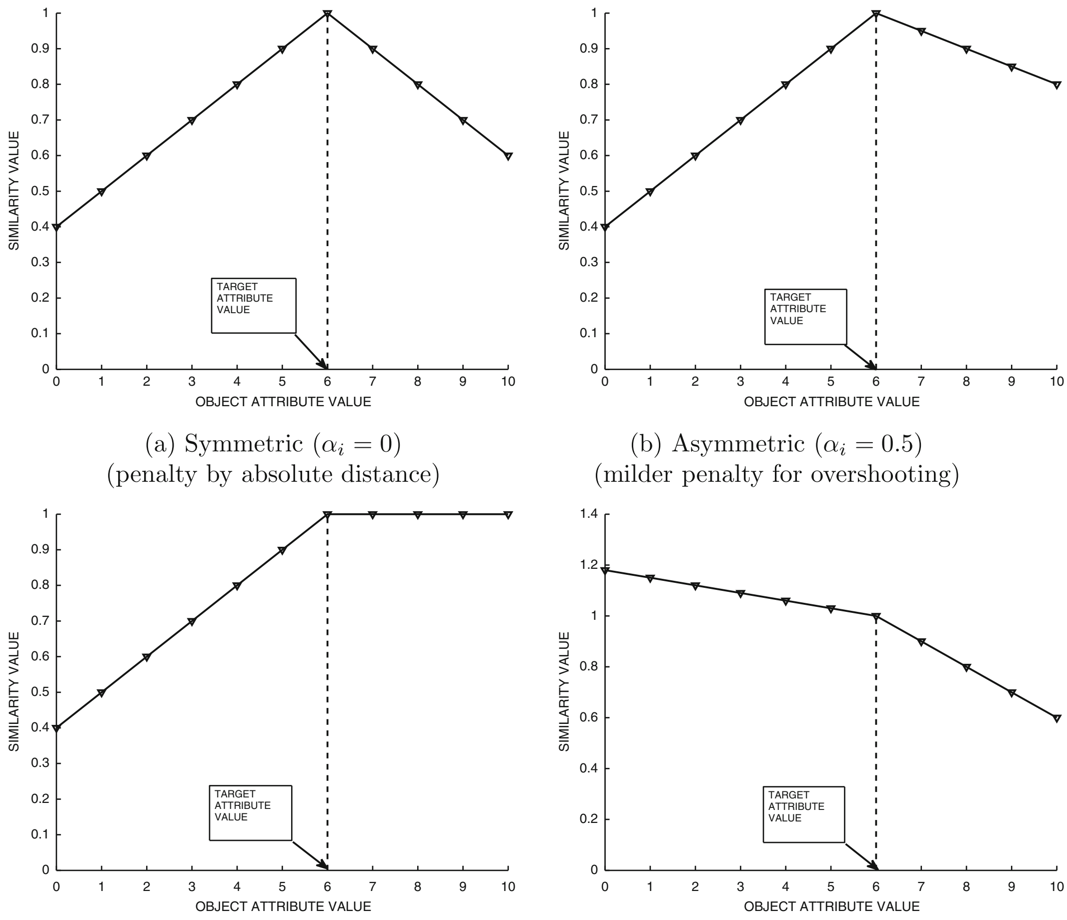
Here, $\alpha_i \geq 0$ is a user-defined parameter, and $I(x_i > t_i)$ is an indicator function that takes on the value of 1 if $x_i > t_i$, and 0 otherwise. Note that the reward kicks in only when the attribute value x_i (e.g., camera resolution) is greater than the target value t_i. For cases in which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

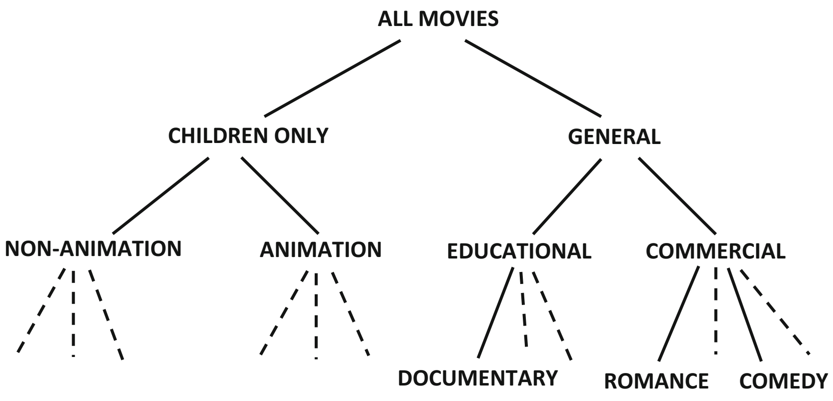


Figure 5.7: An example of hierarchical classification of movie genres

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

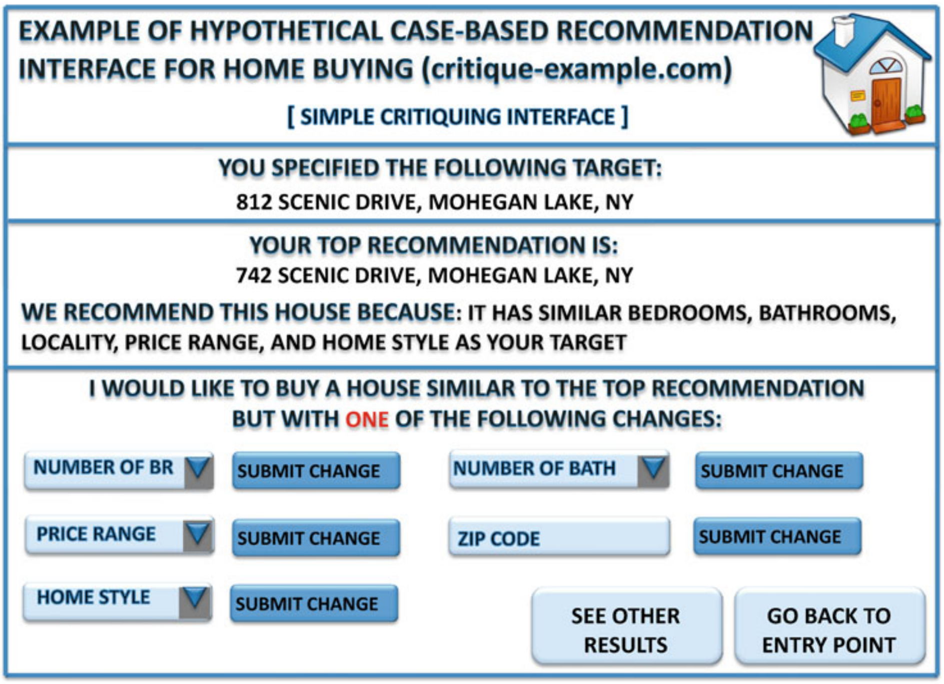
Critiques are motivated by the fact that users are often not in a position to state their requirements exactly in the initial query. In some complex domains, they might even find it difficult to translate their needs in a semantically meaningful way to the attribute values in the product domain. It is 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]

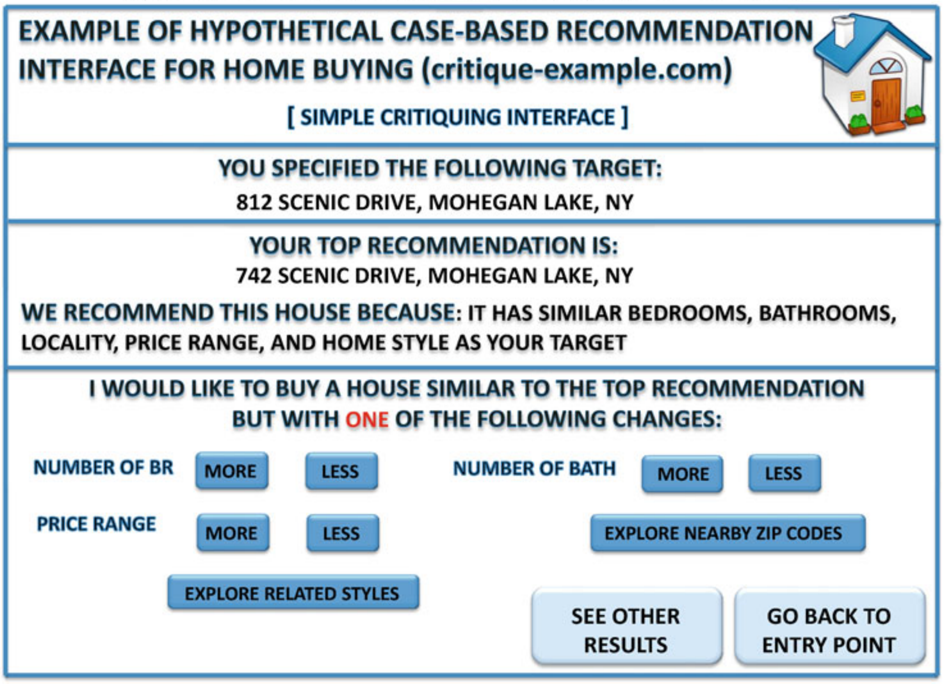
YOU SPECIFIED THE FOLLOWING TARGET:
812 SCENIC DRIVE, MOHEGAN LAKE, NY

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`)

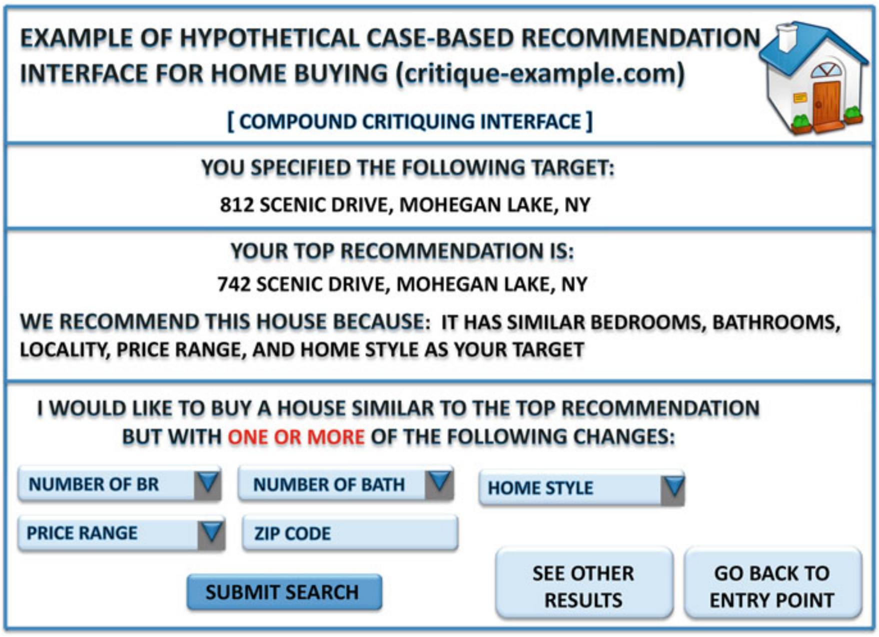
The main problem with the simple critiquing approach is its laborious navigation. If the recommended product contains many features that are required to be changed, then it will lead to a longer chain of subsequent critiques. Furthermore, when one of the features is changed, the recommender system may 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

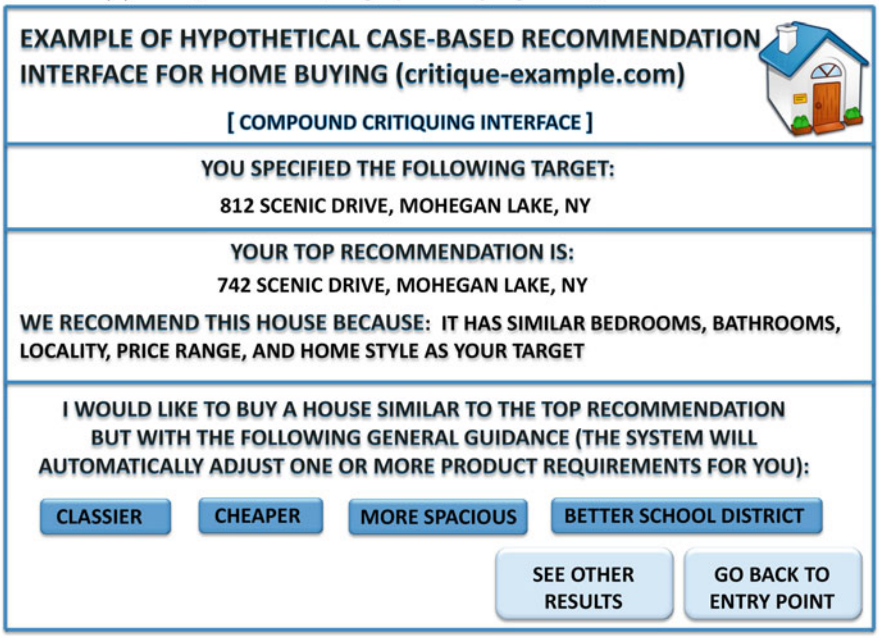
YOUR TOP RECOMMENDATION IS:

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:

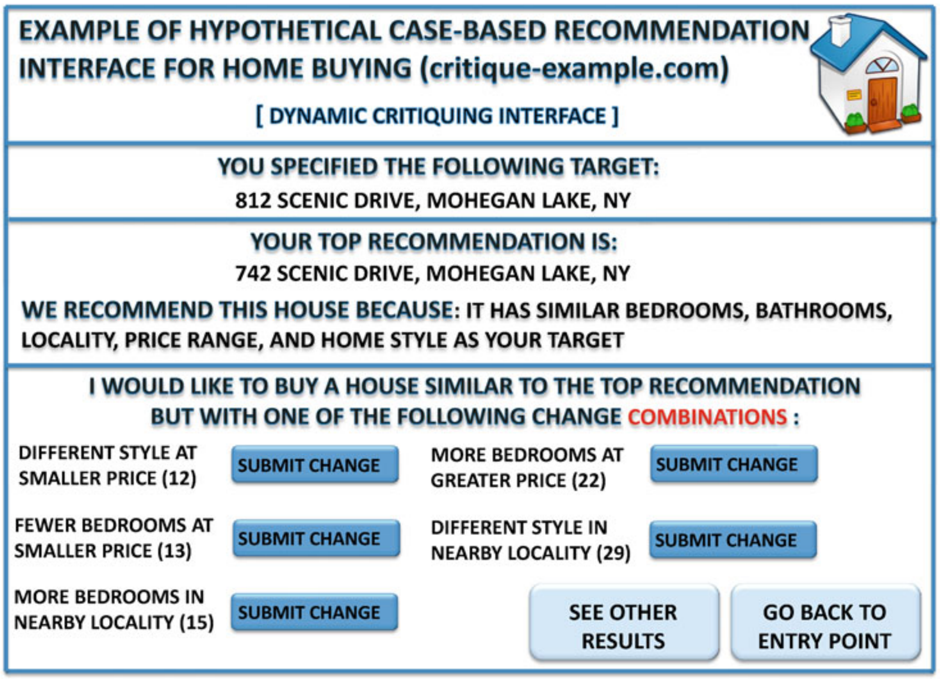
742 SCENIC DRIVE, MOHEGAN LAKE, NY

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

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., *Price* ≤ 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.

Chapter 6

Ensemble-Based and Hybrid Recommender Systems

"*What's better, a poetic intuition or an intellectual work? I think they complement each other.*"– Manuel Puig

6.1 Introduction

In the previous chapters, we discussed three different classes of recommendation methods. Collaborative methods use the ratings of a *community* of users in order to make recommendations, whereas content-based methods use the ratings of a *single* user in conjunction with attribute-centric item descriptions to make recommendations. Knowledge-based methods require the explicit specification of user requirements to make recommendations, and they do not require any historical ratings at all. Therefore, these methods use different sources of data, and they have different strengths and weaknesses. For example, knowledge-based systems can address cold-start issues much better than either content-based or collaborative systems because they do not require ratings. On the other hand, they are weaker than content-based and collaborative systems in terms of using *persistent personalization* from historical data. If a different user enters the same requirements and data in a knowledge-based interactive interface, she might obtain exactly the same result.

All these models seem rather restrictive in isolation, especially when multiple sources of data are available. In general, one would like to make use of all the knowledge available in different data sources and also use the algorithmic power of various recommender systems to make robust inferences. Hybrid recommender systems have been designed to explore these possibilities. There are three primary ways of creating hybrid recommender systems:

C.C. Aggarwal, *Recommender Systems: The Textbook*,
DOI 10.1007/978-3-319-29659-3_6

1. *Ensemble design:* In this design, results from off-the-shelf algorithms are combined into a single and more robust output. For example, one might combine the rating outputs from a content-based and a collaborative recommender into a single output. A significant variation exists in terms of the specific methodologies used for the combination process. The basic principle at work is not very different from the design of ensemble methods in many data mining applications such as clustering, classification, and outlier analysis.

   Ensemble design can be formalized as follows. Let $\hat{R_k}$ be an $m \times n$ matrix containing the *predictions* of the m users for the n items by the kth algorithm, where $k \in \{1 \ldots q\}$. Therefore, a total of q different algorithms are used to arrive at these predictions. The (u, j)th entry of $\hat{R_k}$ contains the predicted rating of user u for item j by the kth algorithm. Note that the observed entries of the original ratings matrix R are replicated in each $\hat{R_k}$, and only the unobserved entries of R vary in different $\hat{R_k}$ because of the different predictions of different algorithms. The final result of the algorithm is obtained by combining the predictions $\hat{R_1} \ldots \hat{R_q}$ into a single output. This combination can be performed in various ways, such as the computation of the weighted average of the various predictions. Furthermore, in some *sequential* ensemble algorithms, the prediction $\hat{R_k}$ may depend on the results of the previous component $\hat{R_{k-1}}$. In yet other cases, the outputs may not be directly combined. Rather, the output of one system is used as an input to the next as a set of content *features*. The common characteristics of all these systems are that (a) they use *existing* recommenders in *off-the-shelf* fashion, and (b) they produce a unified score or ranking.

2. *Monolithic design:* In this case, an integrated recommendation algorithm is created by using various data types. A clear distinction may sometimes not exist between the various parts (e.g., content and collaborative) of the algorithm. In other cases, existing collaborative or content-based recommendation algorithms may need to be modified to be used within the overall approach, even when there are clear distinctions between the content-based and collaborative stages. Therefore, this approach tends to integrate the various data sources more tightly, and one cannot easily view individual components as off-the-shelf black-boxes.

3. *Mixed systems:* Like ensembles, these systems use multiple recommendation algorithms as black-boxes, but the items recommended by the various systems are presented together side by side. For example, the television program for a whole day is a composite entity containing multiple items. It is meaningless to view the recommendation of a single item in isolation; rather, it is the combination of the items that creates the recommendation.

Therefore, the term "hybrid system" is used in a broader context than the term "ensemble system." All ensemble systems are, by definition, hybrid systems, but the converse is not necessarily true.

Although hybrid recommender systems usually combine the power of different types of recommenders (e.g., content- and knowledge-based), there is no reason why such systems cannot combine models of the same type. Since content-based models are essentially text classifiers, it is well known that a wide variety of ensemble models exist to improve the accuracy of classification. Therefore, any classification-based ensemble system can be used to improve the effectiveness of content-based models. This argument also extends to collaborative recommender models. For example, one can easily combine the predicted results of

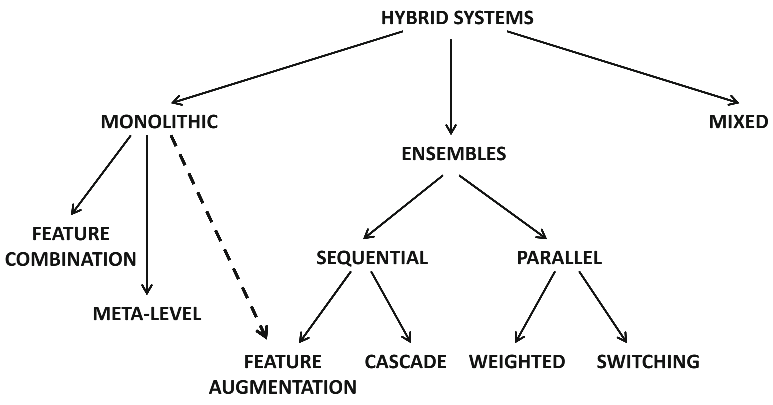


Figure 6.1: The taxonomy of hybrid systems

a latent factor model with those of a neighborhood model to obtain more accurate recommendations [266]. In fact, both[1] the winning entries in the Netflix Prize contest, referred to as "*Bellkor's Pragmatic Chaos*" [311] and "*The Ensemble*" [704], were ensemble systems.

At a broader level, hybrid recommender systems are closely related to the field of ensemble analysis in classification. For example, collaborative models are generalizations of classification models, as discussed in the introduction to Chapter 3. As we will discuss in section 6.2 of this chapter, the theoretical underpinnings of ensemble analysis in classification are similar to those in collaborative filtering. Therefore, this chapter will also focus on how the recommendation approach can be used to improve the effectiveness of collaborative recommender systems in much the same way as one might use ensembles in the field of data classification.

According to Burke [117], hybrid recommender systems can be classified into the following categories:

1. *Weighted:* In this case, the scores of several recommender systems are combined into a single unified score by computing the weighted aggregates of the scores from individual ensemble components. The methodology for weighting the components may be heuristic, or it might use formal statistical models.

2. *Switching:* The algorithm switches between various recommender systems depending on current needs. For example, in earlier phases, one might use a knowledge-based recommender system to avoid cold-start issues. In later phases, when more ratings are available, one might use a content-based or collaborative recommender. Alternatively, the system might adaptively select the specific recommender that provides the most accurate recommendation at a given point in time.

3. *Cascade:* In this case, one recommender system refines the recommendations given by another. In generalized forms of cascades, such as *boosting*, the training process of one recommender system is *biased* by the output of the previous one, and the overall results are combined into a single output.

[1]Both entries were tied on the error rate. The award was given to the former because it was submitted 20 minutes earlier.

4. *Feature augmentation:* The output of one recommender system is used to create input features for the next. While the cascade hybrid successively refines the recommendations of the previous system, the feature augmentation approach treats then as features as *input* for the next system. This approach shares a number of intuitive similarities with the notion of *stacking*, which is commonly used in classification. In stacking, the outputs of one classifier are used as features for the next. Because the different recommenders are (generally) used as off-the-shelf black-boxes, the approach is still an ensemble method (in most cases) rather than a monolithic method.

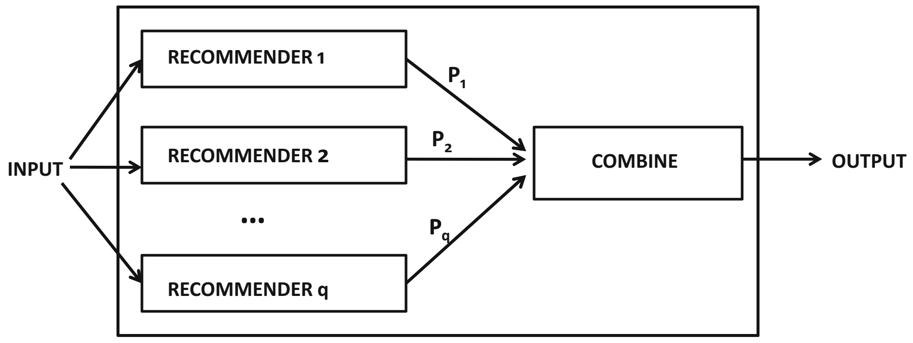
5. *Feature combination:* In this case, the features from different data sources are combined and used in the context of a single recommender system. This approach can be viewed as a monolithic system, and therefore it is not an ensemble method.

6. *Meta-level:* The model used by one recommender system is used as input to another system. The typical combination used is that of a content-based and collaborative system. The collaborative system is modified to use the content features to determine peer groups. Then, the ratings matrix is used in conjunction with this peer group to make predictions. Note that this approach needs to modify the collaborative system to use a content matrix for finding peer groups, although the final predictions are still performed with the ratings matrix. Therefore, the collaborative system needs to be modified, and one cannot use it in an off-the-shelf fashion. This makes the meta-level approach a monolithic system rather than an ensemble system. Some of these methods are also referred to as "collaboration via content" because of the way in which they combine collaborative and content information.

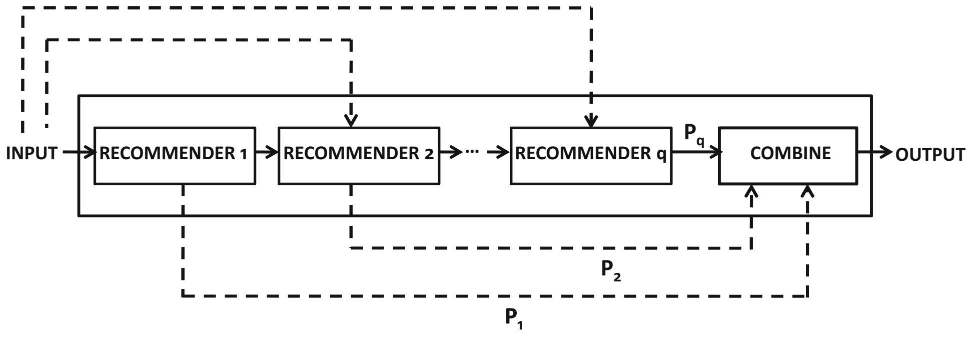
7. *Mixed:* Recommendations from several engines are *presented* to the user at the same time. Strictly speaking, this approach is not an ensemble system, because it does not explicitly combine the scores (of a particular item) from the various components. Furthermore, this approach is often used when the recommendation is a *composite* entity in which multiple items can be recommended as a related set. For example, a composite television program can be constructed from the various recommended items [559]. Therefore, this approach is quite different from all the aforementioned methods. On the one hand, it does use other recommenders as black-boxes (like ensembles), but it does not combine the predicted ratings of the same item from different recommenders. Therefore, mixed recommenders cannot be viewed either as monolithic or ensemble-based methods and are classified into a distinct category of their own. The approach is most relevant in complex item domains, and it is often used in conjunction with knowledge-based recommender systems.

The first four of the aforementioned categories are ensemble systems, the next two are monolithic systems, and the last one is a mixed system. The last category of mixed systems cannot be neatly categorized either as a monolithic or an ensemble system, because it presents multiple recommendations as a composite entity. A hierarchical categorization of these various types of systems is shown in Figure 6.1. Although we have used the higher level categorization of parallel and sequential[2] systems, as introduced by [275], we emphasize that our categorization of Burke's original set of six categories is slightly different from that of [275]. Unlike the taxonomy in [275], which classifies meta-level systems as sequential methods, we view meta-level systems as monolithic because one cannot use off-the-shelf recommendation algorithms, as in the case of a true ensemble. Similarly, the work in [275]

[2]This is also referred to as a *pipelined* system [275].



(a) Parallel design



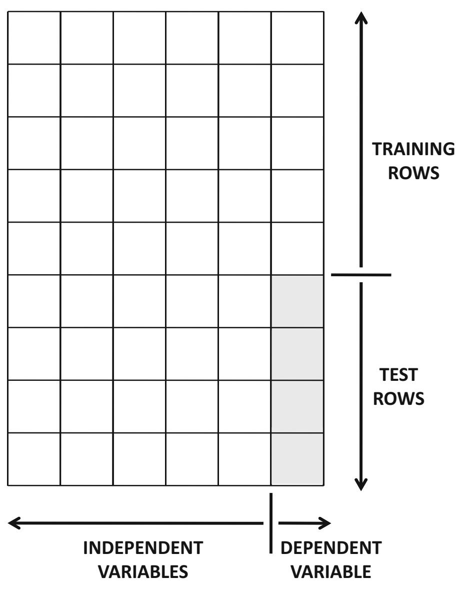
(b) Sequential design

Figure 6.2: Parallel and sequential ensembles

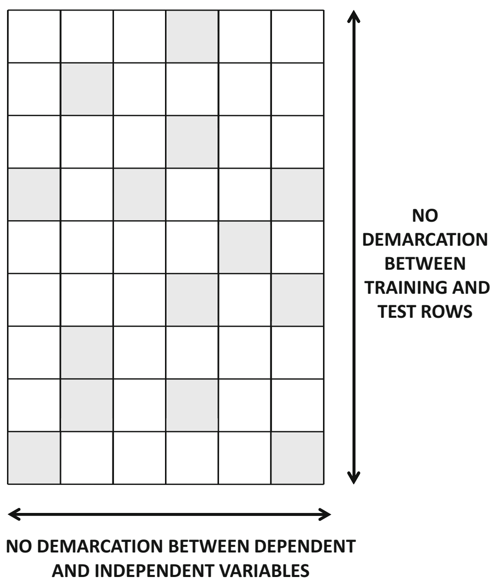
views feature augmentation hybrids as monolithic systems. Although the individual recommenders are combined together in a more complex way in feature augmentation hybrids, the individual recommenders are still used as off-the-shelf black-boxes for the large part. This is the primary distinguishing characteristic of an ensemble system from a monolithic system, and the approach is highly reminiscent of stacking methods in classification. Therefore, we view feature augmentation hybrids as ensemble systems rather than monolithic systems. However, in some cases of feature augmentation hybrids, minor changes are required to the off-the-shelf recommender. In such cases, these systems may be technically considered to have a monolithic design. We have shown this possibility with a dotted line in Figure 6.1.

Aside from the monolithic and mixed methods, which are not truly ensembles, one can view all ensemble methods as having either sequential or parallel designs [275]. In parallel designs, the various recommenders function independently of one another, and the predictions of the individual recommenders are combined at the very end. The weighted and switching methods can be viewed as parallel designs. In sequential designs, the output of one recommender is used as an input to the other. The cascade and meta-level systems can be viewed as examples of sequential methods. A pictorial illustration of the combination process in sequential and parallel systems is shown in Figure 6.2. In this chapter, we will provide a detailed discussion of several recommender systems in each of these categories, although we will use Burke's lower-level taxonomy [117] to organize the discussion.

This chapter is organized as follows. In section 6.2, we discuss the classification perspective of ensemble-based recommender systems. We also explore the level to which the existing theories and methodologies for ensemble methods in the field of classification also



(a) Classification


NO DEMARCATION BETWEEN TRAINING AND TEST ROWS
NO DEMARCATION BETWEEN DEPENDENT AND INDEPENDENT VARIABLES

(b) Collaborative filtering

Figure 6.3: Revisiting Figure 1.4 of Chapter 1. Comparing the traditional classification problem with collaborative filtering. Shaded entries are missing and need to be predicted.

apply to recommender systems. In section 6.3, a number of different examples of weighted hybrids are discussed. In section 6.4, a number of switching hybrids are discussed. Cascade hybrids are discussed in section 6.5, whereas feature augmentation hybrids are discussed in section 6.6. Meta-level hybrids are discussed in section 6.7. Feature combination methods are introduced in section 6.8. Mixed systems are discussed in section 6.9. A summary is given in section 6.10.

6.2 Ensemble Methods from the Classification Perspective

Ensemble methods are commonly used in the field of data classification to improve the robustness of learning algorithms. As we will discuss below, much of this theory also applies to various forms of recommender systems. For example, content-based recommender systems are often straightforward applications of text classification algorithms. Therefore, a direct application of existing ensemble methods in classification is usually sufficient to obtain high-quality results.

As discussed in Chapter 1, collaborative filtering is a generalization of the problem of data classification. We have replicated Figure 1.4 of Chapter 1 in Figure 6.3 to illustrate the relationship between the two problems. It is evident from Figure 6.3(a) that the feature variables and class variable are clearly demarcated in classification. The main distinguishing features of collaborative filtering from classification are that the feature variables and the class variable are not clearly demarcated in the former and that the missing entries may occur in any column or row. The fact that missing entries may occur in any row implies that training and test instances are not clearly demarcated, either. A salient question arises as to whether the bias-variance theory developed in the field of classification [242] also applies to recommender systems. Repeated experiments [266, 311] have shown that combining multiple collaborative recommender systems often leads to more accurate results. This is because the bias-variance theory, which is designed for classification, also applies to the collaborative filtering scenario. This means that many traditional ensemble techniques from classification can also be generalized to collaborative filtering. Nevertheless, because of the fact that the missing entries might occur in any row or column of the data, it is sometimes algorithmically challenging to generalize the ensemble algorithms for classification to collaborative filtering.

We first introduce the bias-variance trade-off as it applies to the field of data classification. Consider a simplified classification or regression model, in which a specific field needs to be predicted, as shown in Figure 6.3(a). It can be shown that the error of a classifier in predicting the dependent variable can be decomposed into three components:

1. *Bias:* Every classifier makes its own modeling assumptions about the nature of the decision boundary between classes. For example, a linear SVM classifier assumes that the two classes may be separated by a linear decision boundary. This is, of course, not true in practice. In other words, any given linear support vector machine will have an inherent *bias*. When a classifier has high bias, it will make *consistently incorrect* predictions over particular choices of test instances near the incorrectly modeled decision-boundary, even when different samples of the training data are used for the learning process.

2. *Variance:* Random variations in the choices of the training data will lead to different models. As a result, the dependent variable for a test instance might be inconsistently predicted by different choices of training data sets. Model variance is closely related to overfitting. When a classifier has an overfitting tendency, it will make *inconsistent* predictions for the same test instance over different training data sets.

3. *Noise:* The noise refers to the intrinsic errors in the target class labeling. Because this is an intrinsic aspect of data quality, there is little that one can do to correct it. Therefore, the focus of ensemble analysis is generally on reducing bias and variance.

The expected mean-squared error of a classifier over a set of test instances can be shown to be sum of the bias, variance, and noise. This relationship can be stated as follows:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Noise} \tag{6.1}$$

It is noteworthy that by reducing either the bias or variance components, one can reduce the overall error of a classifier. For example, classification ensemble methods such as *bagging* [99] reduce the variance, whereas methods such as *boosting* [206] can reduce the bias. It is noteworthy that the only difference between classification and collaborative filtering is that missing entries can occur in any column rather than only in the class variable. Nevertheless, the bias-variance result still holds when applied to the problem of predicting a specific column, whether the other columns are incompletely specified or not. This means that the basic principles of ensemble analysis in classification are also valid for collaborative filtering. Indeed, as we will see later in this chapter, many classical ensemble methods in classification, such as bagging and boosting, have also been adapted to collaborative filtering.

6.3 Weighted Hybrids

Let $R = [r_{uj}]$ be an $m \times n$ ratings matrix. In weighted hybrids, the outputs of various recommender systems are combined using a set of weights. Let $\hat{R}_1 \ldots \hat{R}_q$ be the $m \times n$ *completely specified* ratings matrices, in which the unobserved entries of R are predicted by q different algorithms. Note that the entries r_{uj} that are already observed in the original $m \times n$ ratings matrix R are already fixed to their observed values in each prediction matrix $\hat{R}_k$. Then, for a set of weights $\alpha_1 \ldots \alpha_q$, the weighted hybrid creates a combined prediction matrix $\hat{R} = [\hat{r}_{uj}]$ as follows:

$$\hat{R} = \sum_{i=1}^{q} \alpha_i \hat{R}_i \tag{6.2}$$

In the simplest case, it is possible to choose $\alpha_1 = \alpha_2 = \ldots = \alpha_q = 1/q$. However, it is ideally desired to weight the various systems in a differential way, so as to give greater importance to the more accurate systems. A number of methods exist for such differential weighting. One can also write the aforementioned equation in terms of individual entries of the matrix:

$$\hat{r}_{uj} = \sum_{i=1}^{q} \alpha_i \hat{r}^i_{uj} \tag{6.3}$$

Here $\hat{r}^i_{uj}$ denotes the prediction of the ith ensemble component for user u and item j and $\hat{r}_{uj}$ denotes the final prediction.

In order to determine the optimal weights, it is necessary to be able to evaluate the effectiveness of a particular combination of weights $\alpha_1 \ldots \alpha_q$. While this topic will discussed in more detail in Chapter 7, we will provide a simple evaluation approach here for the purpose of discussion. A simple approach is to hold out a small fraction (e.g., 25%) of the known entries in the $m \times n$ ratings matrix $R = [r_{uj}]$ and create the prediction matrices $\hat{R}_1 \ldots \hat{R}_q$ by applying the q different base algorithms on the remaining 75% of the entries in R. The resulting predictions $\hat{R}_1 \ldots \hat{R}_q$ are then combined to create the ensemble-based prediction $\hat{R}$ according to Equation 6.2. Let the user-item indices (u, j) of these held-out entries be denoted by H. Then, for a given vector $\overline{\alpha} = (\alpha_1 \ldots \alpha_q)$ of weights, the effectiveness of a particular scheme can be evaluated using either the mean-squared error (MSE) or the mean absolute error (MAE) of the predicted matrix $\hat{R} = [\hat{r}_{uj}]_{m \times n}$ over the held-out ratings in H:

$$MSE(\overline{\alpha}) = \frac{\sum_{(u,j)\in H} (\hat{r}_{uj} - r_{uj})^2}{|H|}$$

$$MAE(\overline{\alpha}) = \frac{\sum_{(u,j)\in H} |(\hat{r}_{uj} - r_{uj})|}{|H|}$$

These metrics provide an evaluation of a particular combination of coefficients $\alpha_1 \ldots \alpha_q$. How can we determine the optimal values of $\alpha_1 \ldots \alpha_q$ to minimize these metrics? A simple approach, which works well for the case of MSE, is to use linear regression. It is assumed that the ratings in the held-out set H provide the ground truth values of the dependent variable, and the parameters $\alpha_1 \ldots \alpha_q$ are the independent variables. The idea is to select the independent variables so that the mean-squared error of the linear combination is minimized with respect to the known ratings in the held-out set. The basics of the linear regression model are discussed in section 4.4.5 of Chapter 4, albeit in a different context. The main difference here is in terms of how the dependent and independent variables are defined and in terms of how the linear regression problem is formulated. In this case, the independent variables correspond to the rating predictions of various models for the entry (u, j), and the dependent variable corresponds to the value of each predicted rating $\hat{r}_{uj}$ of the ensemble combination in the held-out set H. Therefore, each observed rating in the held-out set provides a training example for the linear regression model. The regression coefficients correspond to the weights of various component models, and they need to be learned from the (held-out) training examples. After the weights have been learned using linear regression, the individual component models are retrained on the entire training set without any held-out entries. The weights, which were learned using the held-out entries, are used in conjunction with these q models. It is important not to forget this final step in order to ensure that the maximum learning is obtained from all the information available in the ratings. The linear regression approach to model combination is discussed in [266]. A related approach, which can make good use of all the knowledge in the training data, is that of *cross-validation*. Cross-validation methods are discussed in Chapter 7.

Although many systems simply average the results of multiple models, the use of regression is important to ensure that the various models are weighted appropriately. Such regression-based algorithms were included among many of the highly performing entries in the Netflix Prize contest [311, 554], and they are closely related to the concept of stacking in data classification.

The linear regression approach is, however, sensitive to presence of noise and outliers. This is because the squared error function is overly influenced by the largest errors in the data. A variety of *robust regression* methods are available, which are more resistant to the presence of noise and outliers. One such method uses the mean absolute error (MAE) as the objective function as opposed to the mean-squared error. The MAE is well known to be more robust to noise and outliers because it does not overemphasize large errors. A common approach is to use gradient descent method to determine the optimal value of the parameter vector $(\alpha_1 \dots \alpha_q)$ of Equation 6.3. The algorithm starts by setting $\alpha_1 = \alpha_2 = \dots = \alpha_q = 1/q$. Subsequently, the gradient is computed over the held-out entries in H as follows:

$$\frac{\partial MAE(\overline{\alpha})}{\partial \alpha_i} = \frac{\sum_{(u,j)\in H} \frac{\partial |(\hat{r}_{uj} - r_{uj})|}{\partial \alpha_i}}{|H|} \tag{6.4}$$

The value of $\hat{r}_{uj}$ can be expanded using Equation 6.3, and the partial derivative may be simplified in terms of the ratings of individual ensemble components as follows:

$$\frac{\partial MAE(\overline{\alpha})}{\partial \alpha_i} = \frac{\sum_{(u,j)\in H} \text{sign}(\hat{r}_{uj} - r_{uj})\hat{r}^i_{uj}}{|H|} \tag{6.5}$$

The gradient can be written in terms of the individual partial derivatives:

$$\overline{\nabla MAE} = \left(\frac{\partial MAE(\overline{\alpha})}{\partial \alpha_1} \dots \frac{\partial MAE(\overline{\alpha})}{\partial \alpha_q} \right)$$

This gradient is then used to descend through the parameter space $\overline{\alpha}$ with an iterative gradient descent approach as follows:

1. Initialize $\overline{\alpha}^{(0)} = (1/q \dots 1/q)$ and $t = 0$.

2. **Iterative Step 1:** Update $\overline{\alpha}^{(t+1)} \Leftarrow \overline{\alpha}^{(t)} - \gamma \cdot \overline{\nabla MAE}$. The value of $\gamma > 0$ can be determined using a line search so that the maximum improvement in MAE is achieved.

3. **Iterative Step 2:** Update the iteration index as $t \Leftarrow t + 1$.

4. **Iterative Step 3 (convergence check):** If MAE has improved by at least a minimum amount since the last iteration, then go to iterative step 1.

5. Report $\overline{\alpha}^{(t)}$.

Regularization can be added to prevent overfitting. It is also possible to add other constraints on the various values of α_i such as non-negativity or ensuring that they sum to 1. Such natural constraints improve generalizability to unseen entries. The gradient descent equations can be modified relatively easily to respect these constraints. After the optimal weights have been determined, all ensemble models are retrained on the entire ratings matrix without any held-out entries. The predictions of these models are combined with the use of the weight vector discovered by the iterative approach.

There are other ways of performing parameter searches. A simpler approach is to try several judiciously chosen combinations of parameters on a held-out set of ratings. For example, one might tune the various values of α_i in succession by trying different values and holding the others constant. Such an approach is generally applied to various types of parameter tuning [311], and it can often provide reasonably accurate results. Examples of various search techniques are provided in [162, 659].

These methods can be enhanced further with different types of meta-level content features [65, 66, 554]. These methods are discussed in section 6.8.2. Many of the existing ensemble methods do not use these sophisticated combination schemes. Often, these techniques use a simple average of the predictions of different components. It is particularly important to weight the different components when the predicted utility values are on different scales, or when some of the ensemble components are much more accurate than others. In the following, we will provide specific examples of how different types of models are often combined.

6.3.1 Various Types of Model Combinations

In weighted model combinations, a variety of recommendation engines can be combined. There are typically two forms of model combinations:

1. *Homogeneous data type and model classes:* In this case, different models are applied on the same data. For example, one might apply various collaborative filtering engines such as neighborhood-based methods, *SVD*, and Bayes techniques on a ratings matrix. The results are then aggregated into a single predicted value. Such an approach is robust because it avoids the specific bias of particular algorithms on a given data set even though all the constituent models belong to the same class (e.g., collaborative methods). An example of such a blend is provided in [266]. It was shown in [637], how the combination of an ensemble of three different matrix factorization methods can provide high-quality results. In particular, regularized matrix factorization, non-negative matrix factorization, and maximum margin factorization were used as the ensemble components, and the corresponding results were averaged. An interesting *fusion ensemble*, discussed in [67], uses the same recommendation algorithm for various ensemble components, but with different choices of parameters or algorithmic design choices. For example, different numbers of latent factors may be used in an *SVD* algorithm, different numbers of nearest neighbors may be used in a neighborhood-based algorithm, or the choice of the similarity metric may be varied. A simple average

of the predictions of various systems is used. As shown in [67], this simple approach almost always improved the performance of the base model. An earlier variation of this approach [180] uses ensembles of maximum margin matrix factorization methods but with different parameter settings. The work in [338] combines a user-based and item-based neighborhood algorithm.

2. *Heterogeneous data type and model classes:* In this cases, different *classes of* models are applied to different data sources. For example, one component of the model might be a collaborative recommender that uses a ratings matrix, whereas another component of the model might be a content-based recommender. This approach essentially fuses the power of multiple data sources into the combination process. The idea is to leverage the complementary knowledge in the various data sources in order to provide the most accurate recommendations. For example, the work in [659] combines a collaborative and knowledge-based recommender, whereas the work in [162] combines a content-based and collaborative recommender. When working with different data types, it becomes particularly important to carefully weight the predictions of various ensemble components.

These different forms of models provide excellent flexibility in exploring several types of model combinations.

6.3.2 Adapting Bagging from Classification

As discussed earlier in this chapter, the theoretical results on the bias-variance trade-off also hold for the collaborative filtering problem, because the latter problem is a direct generalization of classification. One of the common weighted combination techniques used in the classification problem is that of bagging. Therefore, this method can be used in collaborative filtering as well. However, the bagging approach needs to be slightly modified in order to adjust for the fact that the collaborative filtering problem is formulated somewhat differently from that of classification. First, we discuss bagging in the context of classification.

The basic idea in bagging is to reduce the variance component of the error in classification. In bagging, q training data sets are created with *bootstrapped sampling*. In bootstrapped sampling, rows of the data matrix are sampled *with replacement* in order to create a new training data set of the same size as the original training data set. This new training data set typically contains many duplicates. Furthermore, it can be shown that the expected fraction of rows from the original data matrix that is not represented in a given bootstrapped sample is given by $1/e$, where e is the base of the natural logarithm. A total of q training models are created with each of the sampled training data sets. For a given test instance, the average prediction from these q models is reported. Bagging generally improves the classification accuracy because it reduces the variance component of the error. A particular variant of bagging, known as *subagging* [111, 112], *subsamples* the rows, rather than sampling with replacement. For example, one can simply use all the distinct rows in a bootstrapped sample for training the models. The bagging and subagging methods can be generalized to collaborative filtering as follows:

1. *Row-wise bootstrapping:* In this case, the rows of the ratings matrix R are sampled with replacement to create a new ratings matrix of the same dimensions. A total of q such ratings matrices $R_1 \dots R_q$ are thus created. Note that rows may be duplicated in the process of sampling, although they are treated as separate rows. An existing collaborative filtering algorithm (e.g., latent factor model) is then applied to each of

the q training data sets. For each training data set, an item rating can be predicted for a user only if that user is represented at least once in the matrix. In such a case, the predicted rating from that ensemble component is the average rating[3] of that item over the duplicate occurrences of that user. The predicted rating is then averaged over all the ensemble components in which that user is present. Note that for reasonably large values of q, each user will typically be present in at least one ensemble component with a high probability value of $1 - (1/e)^q$. Therefore, all users will be represented with high probability.

2. *Row-wise subsampling:* This approach is similar to row-wise bootstrapping, except that the rows are sampled without replacement. The fraction f of rows sampled is chosen randomly from $(0.1, 0.5)$. The number of ensemble components q should be significantly greater than 10 to ensure that all rows are represented. The main problem with this approach is that it is difficult to predict all the entries in this setting, and therefore one has to average over a smaller number of components. Therefore, the benefits of variance reduction are not fully achieved.

3. *Entry-wise bagging:* In this case, the *entries* of the original ratings matrix are sampled with replacement to create the q different ratings matrices $R_1 \ldots R_q$. Because many entries may be sampled repeatedly, the entries are now associated with weights. Therefore, a base collaborative filtering algorithm is required that can handle entries with weights. Such algorithms are discussed in section 6.5.2.1. As in the case of row-wise bagging, the predicted ratings are averaged over the various ensemble components.

4. *Entry-wise subsampling:* In entry-wise subsampling, a fraction of the entries are retained at random from the ratings matrix R to create a sampled training data set. Typically, a value of f is sampled from $(0.1, 0.5)$, and then a fraction f of the entries in the original ratings matrix are randomly chosen and retained. This approach is repeated to create q training data sets $R_1 \ldots R_q$. Thus, each user and each item is represented in each subsampled matrix, but the number of specified entries in the subsampled matrix is smaller than that in the original training data. A collaborative filtering algorithm (e.g., latent factor model) is applied to each ratings matrix to create a predicted matrix. The final prediction is the simple average of these q different predictions.

In the aforementioned methods, the final step of the ensemble uses a simple average of the predictions rather than a weighted average. The reason for using a simple average is that all model components are created with an identical probabilistic approach and should therefore be weighted equally. In many of these cases, it is important to choose unstable base methods to achieve good performance gains.

Although the aforementioned discussion provides an overview of the various possibilities for variance reduction, only a small subset of these possibilities have actually been explored and evaluated in the research literature. For example, we are not aware of any experimental results on the effectiveness of subsampling methods. Although subsampling methods often provide superior results to bagging in the classification domain [658], their effect on collaborative filtering is difficult to predict in sparse matrices. In sparse matrices, dropping entries could lead to the inability to predict some users or items at all, which can sometimes worsen the overall performance. A discussion of bagging algorithms in the context of collaborative

[3]It is possible for the unspecified values in duplicate rows to predicted differently, even though this is relatively unusual for most collaborative filtering algorithms.

filtering can be found in [67]. In this work, a row-wise bootstrapping approach is used, and duplicate rows are treated as weighted rows. Therefore, the approach assumes that the base predictor can handle weighted rows. As discussed in [67], significant improvements in the error were achieved with bagging, although the approach seemed to be somewhat sensitive to the choice of base predictor. In particular, according to the results in [67], the bagging approach improved the accuracy over most of the base predictors, with the exception of the factorized neighborhood model [72]. This might possibly be a result of a high level of correlation between the predictions of the various bagged models, when the factorized neighborhood method is used. In general, it is desirable to use uncorrelated base models with low bias and high variance in order to extract the maximum benefit from bagging. In cases where bagging does not work because of high correlations across base predictors, it may be helpful to explicitly use *randomness injection*.

6.3.3 Randomness Injection

Randomness injection is an approach that shares many principles of random forests in classification [22]. The basic idea is to take a base classifier and explicitly inject randomness into the classifier. Various methods can be used for injecting the randomness. Some examples [67] are as follows:

1. *Injecting randomness into a neighborhood model:* Instead of using the top-k nearest neighbors (users or items) in a user-based or item-based neighborhood model, the top-$\alpha \cdot k$ neighbors are selected for $\alpha \gg 1$. Then, k elements are randomly selected from these $\alpha \cdot k$ neighbors. This approach can, however, be shown to be an indirect variant of row-wise subsampling at factor $1/\alpha$. The average prediction from the various components is returned by the approach.

2. *Injecting randomness into a matrix factorization model:* Matrix factorization methods are inherently randomized methods because they perform gradient descent over the solution space after randomly initializing the factor matrices. Therefore, by choosing different initializations, different solutions are often obtained. The combinations of these different solutions often provide more accurate results.

A simple average of the predictions of the different components is returned by the randomized ensemble. Like random forests, this approach can reduce the variance of the ensemble without affecting the bias significantly. In many cases, this approach works quite well where bagging does not work because of a high level of correlation between various predictors. As shown in [67], the randomness injection approach works quite well when the factorized neighborhood model is used as the base predictor [72]. It is noteworthy that the bagging approach does not work very well in the case of the factorized neighborhood model.

6.4 Switching Hybrids

Switching hybrids are used most commonly in recommender systems in the context of the problem of *model selection*, but they are often not formally recognized as hybrid systems. The original motivation for switching systems [117] was to handle the cold-start problem, where a particular model works better in earlier stages when there is a paucity of available data. However, in later stages, a different model is more effective, and therefore one switches to the more effective model.

It is also possible to view switching models in the more general sense of *model selection*. For example, even the parameter selection step of most recommender models requires the running of the model over multiple parameter values and then selecting the optimal one. This particular form of model selection is adapted from the classification literature, and it is also referred to as the *bucket-of-models*. In the following, we discuss both these types of hybrids.

6.4.1 Switching Mechanisms for Cold-Start Issues

Switching mechanisms are often used to handle the cold-start problem, in which one recommender performs better with less data, whereas the other recommender performs better with more data. One might use a knowledge-based recommender, when few ratings are available because knowledge-based recommender systems can function without any ratings, and they are dependent on user specifications of their needs. However, as more ratings become available, one might switch to a collaborative recommender. One can also combine content-based and collaborative recommenders in this way, because content-based recommenders can work well for new items, whereas collaborative recommenders cannot effectively give recommendations for new items.

The work in [85] proposes the *Daily Learner* system in which various recommenders are used in an ordered strategy. If sufficient recommendations are not found by earlier recommenders, then later recommenders are used. In particular, the work in [85] uses two content-based recommenders and a single collaborative recommender. First, a nearest neighbor content classifier is used, followed by a collaborative system, and finally a naive Bayes content classifier is used to match with the long-term profile. This approach does not fully address the cold-start problem because all the underlying learners need some amount of data. Another work [659] combines hybrid versions of collaborative and knowledge-based systems. The knowledge-based system provides more accurate results during the cold-start phase, whereas the collaborative system provides more accurate results in later stages. Incorporating knowledge-based systems is generally more desirable for handling the cold-start problem.

6.4.2 Bucket-of-Models

In this approach, a fraction (e.g., 25% to 33%) of the specified entries in the ratings matrix are held out, and various models are applied to the resulting matrix. The held-out entries are then used to evaluate the effectiveness of the model in terms of a standard measure, such as the MSE or the MAE. The model that yields the lowest MSE or MAE is used as the relevant one. This approach is also commonly used for parameter tuning. For example, each model may correspond to a different value of the parameter of the algorithm, and the value providing the best result is selected as the relevant one. Once the relevant model has been selected, it is retrained on the *entire* ratings matrix, and the results are reported. Instead of using a hold-out approach, a different technique known as *cross-validation* is also used. You will learn more about hold-out and cross-validation techniques in Chapter 7. The bucket-of-models is the single most useful ensemble approach in recommender systems, although it is rarely recognized as an ensemble system unless the different models are derived from heterogeneous data types. When the bucket-of-models is used in the context of a dynamically changing ratings matrix, it is possible for the system to switch from one component to the other. However, when it is used for static data, the system can also be viewed as a special case of weighted recommenders in which the weight of one component is set to 1, and the weights of the remaining components are set to 0.

6.5 Cascade Hybrids

In Burke's original work [117], cascade hybrids were defined in a somewhat narrow way, in which each recommender actively refines the recommendations made by the previous recommender. Here, we take a broader view of cascade hybrids in which a recommender is allowed to use recommendations of the previous recommender in any way (beyond just direct refinement), and then combine the results to make the final recommendation. This broader definition encompasses larger classes of important hybrids, such as boosting, which would not otherwise be included in any of the categories of hybrids. Correspondingly, we define two different categories of cascade recommenders.

6.5.1 Successive Refinement of Recommendations

In this approach, a recommender system successively refines the output of recommendations from the previous iteration. For example, the first recommender can provide a rough ranking and also eliminate many of the potential items. The second level of recommendation then uses this rough ranking to further refine it and break ties. The resulting ranking is presented to the user. An example of such a recommender system is *EntreeC* [117], which uses knowledge of the user's stated interests to provide a rough ranking. The resulting recommendations are then partitioned into buckets of roughly equal preference. The recommendations within a bucket are therefore considered ties at the end of the first stage. A collaborative technique is used to break the ties and rank the recommendations within each bucket. The first knowledge-based recommender is clearly of higher priority because the second-level recommender cannot change the recommendations made at the first level. The other observation is that the second level recommender is much more efficient because it needs to focus only on the ties within each bucket. Therefore, the item-space of each application of the second recommender is much smaller.

6.5.2 Boosting

Boosting has been used popularly in the context of classification [206] and regression [207]. One of the earliest methods for boosting was the *AdaBoost* algorithm [206]. The regression variant of this algorithm is referred to as *AdaBoost.RT* [207]. The regression variant is more relevant to collaborative filtering because it easier to treat ratings as numeric attributes. In traditional boosting, a sequence of training rounds is used with weighted training examples. The weights in each round are modified depending on the performance of the classifier in the previous round. Specifically, the weights on the training examples with error are increased, whereas the weights on the correctly modeled examples are reduced. As a result, the classifier is biased towards correctly classifying the examples that it was unable to properly classify in the previous round. By using several such rounds, one obtains a sequence of classification models. For a given test instance, all models are applied to it, and the weighted prediction is reported as the relevant one.

Boosting needs to be modified to work for collaborative filtering, in which there is no clear demarcation between the training and test rows, and there is also no clear distinction between the dependent and independent columns. A method for modifying boosting for collaborative filtering is proposed in [67]. Unlike classification and regression modeling, in which weights are associated with rows, the training example weights in collaborative filtering are associated with individual *ratings*. Therefore, if the set S represents the set of observed ratings in the training data, then a total of $|S|$ weights are maintained. Note that

S is a set of positions (u, j) in the $m \times n$ ratings matrix R, such that r_{uj} is observed. It is also assumed that the base collaborative filtering algorithm has the capacity to work with weighted ratings (cf. section 6.3). In each iteration, the weights of each of these ratings are modified depending on how well the collaborative filtering algorithm is able to predict that particular entry.

The overall algorithm is applied for a total of T iterations. In the tth iteration, the weight associated with the (u, j)th entry of the ratings matrix is denoted by $W_t(u, j)$. The algorithm starts by equally weighting each entry and predicts all ratings using a baseline model. The prediction of an entry $(u, j) \in S$ is said to be "incorrect," if the predicted rating $\hat{r}_{uj}$ varies from the actual rating r_{uj} by at least a predefined amount δ. The error rate ϵ_t in the tth iteration is computed as the fraction of specified ratings in S for which the predicted value is incorrect, according to this definition. The weights of correctly predicted examples are reduced by multiplying them with ϵ_t, whereas the weights of the incorrectly predicted examples stay unchanged. In each iteration, the weights are always normalized to sum to 1. Therefore, the *relative* weights of incorrectly classified entries always increase across various iterations. The baseline model is applied again to the re-weighted data. This approach is repeated for T iterations, in order to create T different predictions for the unspecified entries. The weighted average of these T different predictions is used as the final prediction of an entry, where the weight of the tth prediction is $\log\left(\frac{1}{\epsilon_t}\right)$. It is noteworthy that the weight update and model combination rules in [67] are slightly different from those used in classification and regression modeling. However, there are very few studies in this area, beyond the work in [67], on using boosting methods for collaborative filtering. It is conceivable that the simple strategies proposed in [67] can be further improved on with experimentation.

6.5.2.1 Weighted Base Models

The boosting and bagging methods require the use of weighted base models, in which entries are associated with weights. In this section, we show how existing collaborative filtering models can be modified so that they can work with weights.

Let us assume that the weight w_{uk} be associated with a particular entry in the ratings matrix for user u and item k. It is relatively straightforward to modify existing models to work with weights on the entries:

1. *Neighborhood-based algorithms:* The average rating of a user is computed in a weighted way for mean-centering the ratings. Both the Pearson and the cosine measures can be modified to take weights into account. Therefore, Equation 2.2 of Chapter 2 can be modified as follows to compute the Pearson coefficient between users u and v:

$$\text{Pearson}(u, v) = \frac{\sum_{k \in I_u \cap I_v} \max\{w_{uk}, w_{vk}\} \cdot (r_{uk} - \mu_u) \cdot (r_{vk} - \mu_v)}{\sqrt{\sum_{k \in I_u \cap I_v} w_{uk}(r_{uk} - \mu_u)^2} \cdot \sqrt{\sum_{k \in I_u \cap I_v} w_{vk}(r_{vk} - \mu_v)^2}} \tag{6.6}$$

The reader should refer to section 2.3 of Chapter 2 for the details of the notations. A different way[4] of modifying the measure is as follows:

$$\text{Pearson}(u, v) = \frac{\sum_{k \in I_u \cap I_v} w_{uk} \cdot w_{vk} \cdot (r_{uk} - \mu_u) \cdot (r_{vk} - \mu_v)}{\sqrt{\sum_{k \in I_u \cap I_v} w_{uk}^2(r_{uk} - \mu_u)^2} \cdot \sqrt{\sum_{k \in I_u \cap I_v} w_{vk}^2(r_{vk} - \mu_v)^2}} \tag{6.7}$$

[4]The work in [67] proposes only the first technique for computing the similarity.

For item-item similarity measures, the adjusted cosine measure can be modified in a similar way. These weighted similarity measures are used both for computing the nearest neighbors and for (weighted) averaging of the ratings in the peer group.

2. *Latent factor models:* Latent factor models are defined as optimization problems in which the sum of the squares of the errors of the specified entries are minimized. In this case, the *weighted* sum of the squares of the optimization problem must be minimized. Therefore, the objective function in section 3.6.4.2 of Chapter 3 can be modified as follows:

$$\text{Minimize } J = \frac{1}{2} \sum_{(i,j) \in S} w_{ij} e_{ij}^2 + \frac{\lambda}{2} \sum_{i=1}^{m} \sum_{s=1}^{k} u_{is}^2 + \frac{\lambda}{2} \sum_{j=1}^{n} \sum_{s=1}^{k} v_{js}^2 \qquad (6.8)$$

Here, $U = [u_{ij}]$ and $V = [v_{ij}]$ are the $m \times k$ and $n \times k$ user-factor and item-factor matrices, respectively. Note the weights associated with the errors on the entries. The corresponding change in the gradient descent method is to weight the relevant updates:

$$u_{iq} \Leftarrow u_{iq} + \alpha(w_{ij} \cdot e_{ij} \cdot v_{jq} - \lambda \cdot u_{iq})$$
$$v_{jq} \Leftarrow v_{jq} + \alpha(w_{ij} \cdot e_{ij} \cdot u_{iq} - \lambda \cdot v_{jq})$$

Many other base collaborative filtering algorithms can be modified to work with weighted entries. These types of weighted base algorithms are useful for many collaborative filtering ensembles, such as boosting and bagging.

6.6 Feature Augmentation Hybrids

The feature augmentation hybrid shares a number of intuitive similarities with the stacking ensemble in classification. In stacking [634], the first level classifier is used to create or augment a set of features for the second level classifier. In many cases, off-the-shelf systems are used like an ensemble. However, in some cases, changes may be required to the component recommender system to work with the modified data, and therefore the hybrid system is not a true ensemble of off-the-shelf systems.

The *Libra* system [448] combines Amazon.com's recommender system with its own Bayes classifier. The approach uses the "related authors" and "related titles" that Amazon generates as features describing the items. Note that Amazon generates these recommendations with the use of a collaborative recommender system. These data are then used in conjunction with a content-based recommender to make the final predictions. Note that any off-the-shelf content-based system can be used in principle, and therefore the approach can be viewed as an ensemble system. The approach in [448] opts for a naive Bayes text classifier. It was found through experiments that the features generated by Amazon's collaborative system were of high quality, and they contributed significantly to better quality recommendations.

Instead of using a collaborative system first, it is also possible to use the content-based system first. The basic idea is to use the content-based system to fill in the missing entries of the ratings matrix so that it is no longer sparse. Thus, the missing entries are estimated by the content-based system to create a denser ratings matrix. These newly added ratings are referred to as *pseudo-ratings.* Then, a collaborative recommender is used on the dense ratings matrix to make rating predictions. Finally, the collaborative prediction is combined with the original content-based prediction in a weighted way to yield the overall prediction of the missing entries in the matrix [431]. The incorporation of missing ratings in the first phase allows for a more robust application of the second phase in terms of similarity computation. However, the similarity computation does need to be modified to give less weight to pseudo-ratings compared to true ratings. This is because pseudo-ratings were inferred, and they might be error-prone.

How can such weights be determined? The weight of a pseudo-rating intuitively represents the algorithm's certainty in the prediction of the first phase, and it is an increasing function of the number of ratings $|I_i|$ of that user. A number of heuristic functions are used to weight various ratings, and the reader is referred to [431] for details. Note that this approach requires modifications to the second phase of collaborative filtering, and off-the-shelf algorithms cannot be used. Such methods can be viewed as monolithic systems.

Feature augmentation has a long history in recommender systems. One of the earliest example of feature augmentation was implemented in the context of the *GroupLens* system [526], in which a knowledge-based system was used to create a database of artificial ratings. The agents, known as filterbots, used specific criteria such as the number of spelling errors or the message size to assign ratings to items, while acting as artificial users. Subsequently, these ratings were used in the context of a collaborative system to make recommendations.

6.7 Meta-Level Hybrids

In a meta-level hybrid, the *model* learned by one recommender is used as input to the next level. An important example of collaboration via content was the early work by Pazzani [475]. A content-based model [363] is constructed that describes the discriminative features predicting restaurants. The discriminative features may be determined using any of the feature selection methods discussed in section 4.3 of Chapter 4. Each user is defined by a vector representation of discriminative words. An example of the possible user-word matrix for a restaurant recommender systems is shown below:

Word $\Rightarrow$ **User** $\Downarrow$	beef	roasted	lamb	fried	eggs
Sayani	0	3	0	2.5	1.7
John	2.3	1.3	0.2	1.4	2.1
Mary	0	2.8	0.9	1.1	2.6
Peter	2.4	1.7	0	3.5	1.9
Jack	1.6	2.2	3.1	1.0	0

The weights in the aforementioned table may be obtained using the descriptions of the items that the user has accessed. Note that the irrelevant words have already been removed because the content-based feature selection in the first phase creates a discriminative vector-space representation for each user. Furthermore, the representation is significantly denser than a typical ratings matrix. Therefore, one can robustly compute the similarities between users with this new representation. The main idea here is that the *content-based* peer group is used to determine the most similar users of the target user. Once the peer group has been determined, then the weighted average of the ratings of the peer group are used to determine the predicted ratings. Note that this approach does require a certain amount of change to the original collaborative recommender, at least in terms of how the similarity is computed. The peer group formation must use the user-word matrix (which was the model created in the first phase), whereas the final recommendation uses the ratings matrix. This is different from a collaborative system in which both stages use the same matrix. Furthermore, the first phase of the approach cannot use off-the-shelf content-based models in their entirety because it is mostly a feature selection (preprocessing) phase. Therefore, in many cases, these systems cannot be considered true ensembles, because they do not use existing methods as off-the-shelf recommenders.

Another example of a meta-level system was *LaboUr* [534] in which an instance-based model is to used to learn the content-based user's profile. The profiles are then compared using a collaborative approach. These models are compared across users to make predictions. Many of these methods fall within the category of "collaboration via content," though that is not the only way in which such hybrids can be constructed.

6.8 Feature Combination Hybrids

In feature combination hybrids, the idea is to combine the input data from various sources (e.g., content and collaborative) into a unified representation before applying a predictive algorithm. In most cases, this predictive algorithm is a content-based algorithm that uses collaborative information as additional features. An example of such an approach was presented in [69], where the *RIPPER* classifier was applied to the augmented data set. It was shown in [69] that the methodology achieved significant improvements over a purely collaborative approach. However, the content features need to be hand picked in order to achieve this result. Therefore, the approach can be sensitive to the choice of data set and feature representation. The approach reduces the sensitivity of the system to the number of users that have rated an item. This is, of course, the property of any content-based system, which is robust to the cold-start problem from the perspective of new items.

Note that it is possible for the combination to be performed in a variety of different ways with different types of background knowledge. For example, consider the case where each item is associated with a higher-level taxonomy representing the genres of the items. The representation profile of the user and items can be augmented in terms of the relevant genres in the hierarchy. The ratings matrix can then be constructed in terms of genres rather than items. In sparse matrices, such an approach can provide more effective results because it reduces the number of columns, and because most entries are likely to be populated in the compressed matrix.

Another approach is to augment a ratings matrix and add columns for keywords in addition to items. Therefore, the ratings matrix becomes an $m \times (n + d)$ matrix, where n is the number of items and d is the number of keywords. The weights of "keyword items" are based on the weighted aggregation of the descriptions of the items accessed, bought, or rated by the user. A traditional neighborhood or matrix factorization approach can be used with this augmented matrix. The relative weights of the two types of columns can be learned through cross-validation (see Chapter 7). This type of combination of two optimization models is common in hybrid settings, where the objective function is set up as follows in terms of a parameter vector $\overline{\theta}$:

$$J = \text{CollaborativeObjective}(\overline{\theta}) + \beta\,\text{ContentObjective}(\overline{\theta}) + \text{Regularization} \qquad (6.9)$$

The objective function is then optimized over the parameter vector $\overline{\theta}$. A specific example, which is discussed below, is the generalization of sparse linear models (cf. section 2.6.5 of Chapter 2) with side information.

6.8.1 Regression and Matrix Factorization

Let R be an $m \times n$ implicit feedback ratings matrix, and C be a $d \times n$ content matrix, in which each item is described by non-negative frequencies of d words. Examples include descriptions of items or short reviews of items. Since R is an implicit feedback matrix, missing entries are assumed to be 0s. As in section 2.6.5, let W be an $n \times n$ item-item coefficient matrix in which the ratings are predicted as $\hat{R} = RW$. However, in this case we can also predict the ratings as $\hat{R} = CW$. Therefore, instead of optimizing only $||R - RW||^2$, we add an additional content-based term $||R - CW||^2$. Together with elastic-net regularization, and non-negativity/diagonal constraints, the enhanced optimization model is stated as follows [456]:

$$\begin{aligned} \text{Minimize } J = ||R - RW|||^2 + \beta \cdot ||R - CW||^2 + \lambda ||W||^2 + \lambda_1 \cdot ||W||_1 \\ \text{subject to:} \\ W \geq 0 \\ \text{Diagonal}(W) = 0 \end{aligned}$$

The weight parameter β can be determined in a tuning phase. Although the ratings can be predicted either as $\hat{R} = RW$ or as $\hat{R} = CW$, only the former prediction function is used. Therefore, the term $||R - CW||^2$ is used only to refine the objective function as an additional regularizer. In other words, the goal of the additional term is to improve the *generalization power* of the model for predicting future (and as yet unknown) actions of the user. Some variations of this basic objective function are discussed in [456].

This type of approach can be used for combining any other type of collaborative filtering (optimization) model with content-based methods. For example, in the case of matrix factorization, one can use an $m \times k$ user factor matrix U, an $n \times k$ *shared* item factor matrix V, and a $d \times k$ content factor matrix Z to set up the optimization model as follows [557]:

$$\text{Minimize } J = ||R - UV^T|||^2 + \beta \cdot ||C - ZV^T||^2 + \lambda(||U||^2 + ||V||^2 + ||Z||^2)$$

Note that the item factor matrix V is shared between the factorizations of the ratings matrix and content matrix. Such shared matrix factorization models are also used for incorporating other types of side information such as social trust data (cf. section 11.3.8 of Chapter 11). An overview of combining matrix factorization methods with arbitrary models is provided in section 3.7.7 of Chapter 3.

6.8.2 Meta-level Features

It is not necessary to use feature combination in the context of multiple types of recommenders (e.g., content and collaborative). New meta-features can be extracted from features of a particular type of recommender and then combined within the ensemble model. For example, one can extract *meta-level features* from a ratings matrix corresponding to the number of ratings given by various users and items. When a user rates many movies, or when a movie is rated by many users, it affects the recommendation accuracy of the various algorithms in different ways. Different recommender systems will be more or less sensitive to these characteristics, and will therefore do better or worse for various users and items. The basic idea of meta-level features is to account for these *entry-specific* differences in the model combination process with the use of meta-features. The resulting meta-features can be paired with other ensemble algorithms to create an ensemble design, which incorporates characteristics from various types of hybrids, but it does not neatly fall into any of Burke's seven original categories [117]. However, it is most closely related to feature combination hybrids in the sense that it combines meta-features with ratings.

Id.	Description
1	Constant value of 1 (using only this feature amounts to using the global linear regression model of section 6.3)
2	A binary variable indicating whether the user rated more than 3 movies on this particular date
3	The log of the number of times a movie has been rated
4	The log of the number of distinct dates on which a user has rated movies
5	A Bayesian estimate of the mean rating of the movie after having subtracted out the user's Bayesian estimated mean
6	The log of the number of user ratings
16	The standard deviation of the user ratings
17	The standard deviation of the movie ratings
18	The log of (Rating Date − First User Rating Date +1)
19	The log of the number of user ratings on the date +1

Table 6.1: A subset of the meta-features used in [554] for ensemble combination on the Netflix Prize data set

The meta-feature approach has proven to be a potentially powerful method for robust ensemble design. In fact, both winning entries in the Netflix Prize content, corresponding to *Bellkor's Pragmatic Chaos* [311] and *The Ensemble* [704], used such an approach. We will describe the use of such meta-level features in collaborative filtering algorithms. In particular, we will discuss the methodology of *feature weighted linear stacking* [554], which combines such meta-level features with the stacking methods discussed earlier in section 6.3. This approach is based on the blending technique used in *The Ensemble* [704]. A subset of the meta-features used in [554] for the stacking process on the Netflix Prize data set is provided in Table 6.1 for illustrative purposes. The identifier in the left column corresponds to the identifier used in the original paper [554]. These features are particularly instructive because one can usually extract analogous features for other ratings data sets. Note that each feature in Table 6.1 is specific to an entry in the ratings matrix.

Let us assume that a total of l (numeric) meta-features have been extracted, and their values are $z_1^{ut} \dots z_l^{ut}$ for user-item pair (u, t). Therefore, the meta-features are specific to each entry (u, t) in the ratings matrix, although some features may take on the same values for varying values of u or varying values of t. For example, feature 3 in Table 6.1 will not vary with the user u, but it will vary with the item t.

Let us assume that there are a total of q base recommendation methods, and the weights associated with the q recommendation methods are denoted by $w_1 \dots w_q$. Then, for a given entry (u, t) in the ratings matrix, if the predictions of the q components are $\hat{r}_{ut}^1 \dots \hat{r}_{ut}^q$, then the prediction $\hat{r}_{ut}$ of the overall ensemble is given by the following:

$$\hat{r}_{ut} = \sum_{i=1}^{q} w_i \hat{r}_{ut}^i \tag{6.10}$$

We would like the estimated prediction $\hat{r}_{ut}$ of the ensemble to match the observed ratings r_{ut} as closely as possible. Note that the approach in section 6.3 uses a linear regression model to learn the weights $w_1 \dots w_q$ by holding out a pre-defined fraction of the entries during the process of training the q models, and then using the held-out entries as the observed values in the linear regression model. Such an approach is pure stacking, and it can be considered a weighted hybrid. However, it can be enhanced further using meta-features. The main idea is that *the linear regression weights $w_1 \ldots w_q$ are specific to each entry in the ratings matrix and they are themselves linear functions of the meta-features.* In other words, the weights now need to be super-scripted with (u, t) to account for the fact that they are specific to each entry (u, t) in the ratings matrix:

$$\hat{r}_{ut} = \sum_{i=1}^{q} w_i^{ut} \hat{r}_{ut}^i \tag{6.11}$$

This is a more refined model because the nature of the combination is local to each entry in the ratings matrix, and it is not blindly global to the entire matrix. The problem is that the number, $m \times n \times q$, of different parameters w_i^{ut} becomes too large to be learned in a robust way. In fact, the number of parameters (weights) is larger than the number of observed ratings, as a result of which overfitting will occur. Therefore, the weights are assumed to be linear combinations of the meta-features under the assumption that these meta-features regulate the relative importance of the various models for the individual user-item combinations. Therefore, we introduce the parameters v_{ij} that regulate the importance of the jth meta-feature to the ith model. The weights for entry (u, t) can now be expressed as linear combination of the meta-feature values of entry (u, t) as follows:

$$w_i^{ut} = \sum_{j=1}^{l} v_{ij} z_j^{ut} \tag{6.12}$$

We can now express the regression modeling problem in terms of a fewer number, $q \times l$, of parameters v_{ij}, where v_{ij} regulates the impact of the jth meta-feature on the relative importance of the ith ensemble model. Substituting the value of w_i^{ut} from Equation 6.12 in Equation 6.11, we obtain the relationship between the ensemble rating and the component ratings as follows:

$$\hat{r}_{ut} = \sum_{i=1}^{q} \sum_{j=1}^{l} v_{ij} z_j^{ut} \hat{r}_{ut}^i \tag{6.13}$$

Note that this is still a linear regression problem in $q \times l$ coefficients corresponding to the variables v_{ij}. A standard least-squares regression model can be used to learn the values of v_{ij} on the held-out[5] ratings. The independent variables of this regression are given by the quantities $z_j^{ut} \hat{r}_{ut}^i$. Regularization can be used to reduce overfitting. After the weights have been learned using linear regression, the individual component models are retrained on the entire training set without any held-out entries. The weights, which were learned using the held-out entries, are used in conjunction with these q models.

6.9 Mixed Hybrids

The main characteristic of mixed recommender systems is that they combine the scores from different components in terms of *presentation*, rather than in terms of combining the predicted scores. In many cases, the recommended items are presented next to one another [121, 623]. Therefore, the main distinguishing characteristic of such systems is the combination of presentation rather than the combination of predicted scores.

[5]In the context of the Netflix Prize contest, this was achieved on a special part of the data set, referred to as the *probe set*. The probe set was not used for building the component ensemble models.

Most of the other hybrid systems focus on creating a unified rating, which is extracted from the various systems. A classical example is illustrated in [559], in which a personalized television listing is created using a mixed system. Typically, a *composite* program is presented to the user. This composite program is created by combining items recommended by different systems. Such composite programs are typical in the use of mixed systems, although the applicability of mixed systems goes beyond such scenarios. In many of these cases, the basic idea is that the recommendation is designed for a relatively complex item containing many components, and it is not meaningful to recommend the individual items. The new item startup problem is often alleviated with a mixed recommender system. Because a television program has many slots, either the content-based or collaborative recommender might be successful in filling the different slots. In some cases, a sufficient number of recommendations for the slots may be achieved only with multiple recommenders of different types, especially at the very beginning when there is a paucity in the available data. However, conflict resolution may be required in some cases where more choices are available than the available slots.

Another example of a mixed hybrid has been proposed in the tourism domain [660, 661]. In this case, bundles of recommendations are created, where each bundle contains multiple categories of items. For example, in a tourism recommender system, the different categories may correspond to accommodations, leisure activities, air-tickets, and so on. Tourists will typically buy bundles of these items from various categories in order to create their trips. For each category, a different recommender system is employed. The basic idea here is that the recommender system that is most appropriate for obtaining the best accommodations, may not be the one that is most appropriate for recommending tourism activities. Therefore, each of these different aspects is treated as a different category for which a different recommender system is employed. Furthermore, it is important to recommend bundles in which the items from multiple categories are not mutually inconsistent. For example, if a tourist is recommended a leisure activity that is very far away from her place of accommodation, then the overall recommendation bundle will not be very convenient for the tourist. Therefore, a knowledge base containing a set of domain constraints is incorporated for the bundling process. The constraints are deigned to resolve inconsistencies in the product domain. A constraint satisfaction problem is employed to determine a mutually consistent bundle. More details of the approach are discussed in [660, 661].

It is noteworthy that many of the mixed hybrids are often used in conjunction with knowledge-based recommender systems as one of the components [121, 660]. This is not a coincidence. Mixed hybrids are generally designed for complex product domains with multiple components like knowledge-based recommender systems.

6.10 Summary

Hybrid recommender systems are used either to leverage the power of multiple data sources or to improve the performance of existing recommender systems within a particular data modality. An important motivation for the construction of hybrid recommender systems is that different types of recommender systems, such as collaborative, content-based, and knowledge-based methods, have different strengths and weaknesses. Some recommender systems work more effectively at cold start, whereas other work more effectively when sufficient data are available. Hybrid recommender systems attempt to leverage the complementary strengths of these systems to create a system with greater overall robustness.

Ensemble methods are also used to improve the accuracy of collaborative filtering methods in which the different components use the same ratings matrix. In these cases, the individual models use the same base data rather than different sources of data. Such methods are much closer to the existing ideas on ensemble analysis in the classification domain. The basic idea is to use the various models to incorporate diversity and reduce model bias. Many of the existing theoretical results on the bias-variance trade-off in classification are also applicable to collaborative filtering applications. Therefore, many techniques, such as bagging and boosting, can be adapted with relatively minor modifications.

Hybrid systems are designed as monolithic systems, ensemble systems, or mixed systems. Ensemble systems are typically designed by using either sequential or parallel arrangement of recommenders. In monolithic design, either existing recommenders are modified, or entirely new recommenders are created by combining the features from multiple data modalities. In mixed systems, recommendations from multiple engines are presented simultaneously. In many cases, meta-features can also be extracted from a particular data modality in order to combine the predictions of various recommenders in an entry-specific way. The great strength of hybrid and ensemble systems arises from their ability to leverage complementary strengths in various systems. The top entries in the Netflix Prize contest were all ensemble systems.

6.11 Bibliographic Notes

Although hybrid systems have a long and rich history in the development of recommender systems, a formal categorization of these methods was not performed until the survey by Burke [117]. A discussion of hybrid recommender systems in the specific context of the Web is provided in [118]. Burke originally categorized recommender systems into seven different categories. Subsequently, Jannach *et al.* [275] created a higher-level categorization of these lower-level categories into pipelined and parallel systems. The hierarchical taxonomy in this book roughly follows the work of [275] and [117], although it makes a number of modifications to include several important methods, such as boosting, into one of these categories. It is important to note that this taxonomy is not exhaustive because many ensemble systems, such as those winning the Netflix Prize, use ideas from many types of hybrids. Nevertheless, Burke's original categorization is very instructive, because it covers most of the important building blocks. Recently, ensemble methods have received a lot of attention, especially after the winning entries in the Netflix Prize contest were both ensemble systems [311, 704].

Ensemble methods have been used extensively in the classification literature. A detailed discussion of the bias-variance trade-off in the context of the classification problem is provided in [22]. Bagging and subsampling methods for classification are discussed in [111–113]. A recent work [67] shows how one might leverage ensemble methods from the classification literature to recommender systems by adapting methods such as bagging and *AdaBoost.RT*. While some ensemble systems are developed with this motivation, other systems combine the power of different data types. Weighted models are among the most popular classes of models. Some of the models combine models built on homogeneous data types. Methods for constructing homogeneous weighted ensembles are discussed in [67, 266]. The winners [311, 704] of the Netflix Prize contest also used a weighted ensemble system, although the combination uses additional meta-features, which imbues it with some of the properties of a feature combination approach. The work in [180] uses ensembles of maximum margin matrix factorization methods with different parameter settings. User-based and item-based neighborhood algorithms are combined in [338]. Other recent work on weighted models shows how to combine systems built on top of different data types. The work in [659] combines a collaborative and knowledge-based recommender, whereas the work in [162] combines a content-based and collaborative recommender.

A performance-based switching hybrid is discussed in [601]. An interesting machine-learning approach to switching mechanisms is discussed in [610]. Other switching mechanisms for handling cold-start issues are discussed in [85]. Another combination of a knowledge-based and collaborative system to create a switching hybrid is discussed in [659].

Cascade systems use sequential processing of the ratings to make recommendations. Such systems can either use refinements or they can use boosting methods. The *EntreeC* recommender [117] is the most well-known example of a cascade system that uses refinements. A cascade system that uses boosting is discussed in [67]. The latter methods uses a weighted version of the *AdaBoost.RT* algorithm in order to create the hybrid recommender.

Feature augmentation hybrids use the recommenders of one type to augment the features of another. The *Libra* system [448] combines Amazon.com's recommender system with its own Bayes classifier. The output of the Amazon system is used to create a content-based recommender. The method in [431] uses a content-based system to estimate the missing entries of the ratings matrix and uses the estimated values in the context of a collaborative system. In the *GroupLens* system [526], a knowledge-based system was used to create a database of artificial ratings. These ratings were used in the context of a collaborative system to make recommendations. The work in [600] shows how to use a feature augmentation hybrid to recommend research papers.

Many techniques have been used recently to create fused feature spaces or unified representations from ratings matrices and content matrices. This unified representation or feature space forms the basis on which machine learning tools can be applied. One of the earliest works along this line constructs joint feature maps [68] from rating and content information and then uses machine learning models in order to perform the prediction. A tensor-based approach is used to achieve this goal. An analogous approach is also used in [557], which jointly factorizes the user-item purchase profile matrix and the item-feature content matrix into a common latent space. This latent representation is then used for learning. The work in [411] uses a latent factor model in which the review text is combined with ratings. A regression-based latent factor model is proposed in [14] for rating prediction, which uses content features for factor estimation. The user and item latent factors are estimated through independent regression on user and item features. Then, a multiplicative function is used on the user and item factors for prediction. Sparse regression models have also been used for fused prediction in [456]. Finally, graph-based models have been used to create unified representations. The work in [238] leans the interaction weights between user actions and various features such as user-item profile information and side information. Unified Boltzmann machines are used to perform the prediction. A unified graph-based representation has been proposed in [129]. A Bayesian network is created containing item nodes, user nodes, and item feature nodes. This Bayesian network is used to perform combined content-based and collaborative recommendations.

In a meta-level hybrid, the *model* learned by one recommender is used as input to the next level. In the early work by Pazzani [475], a content-based model [363] is constructed that describes the discriminative features predicting restaurants. Each user is defined by a vector representation of discriminative words. The content-based model is used to determine the peer group, which is then used for the purpose of recommendation. Meta-level combinations of content-based and collaborative systems are discussed in [475, 534]. A two-stage Bayesian meta-level hybrid is discussed in [166]. A different type of hierarchical Bayes model that combines collaborative and content-based systems is presented in [652]. Methods for stacking recommender systems with meta-features are discussed in [65, 66, 311, 554]. The *STREAM* system [65, 66] was one of the earliest systems to leverage meta-level features.

A number of mixed recommender systems have been proposed in [121, 559, 623, 660, 661]. A mixed recommender system for creating television programs is discussed in [559], whereas a system for providing tourism bundles is discussed in [660]. It is noteworthy that many mixed recommender systems are used in complex product domains like knowledge-based recommender systems [121, 660].

6.12 Exercises

1. How does the rank of the latent factor model affect the bias-variance trade-off in a recommender system? If you had to use a latent factor model as the base model for a bagging ensemble, would you choose a model with high rank or low rank?

2. Does your answer to Exercise 1 change if you had to use boosting in conjunction with a latent factor model?

3. Implement an entry-wise bagging model by using a weighted latent factor model as the base model.

4. Suppose that you created a collaborative system in which the user-item matrix contained word frequencies as additional rows of the matrix. Each additional row is a word, and the value of the word-item combination is a frequency. An item-based neighborhood model is used with this augmented representation. What kind of hybrid would this be considered? Discuss the possible impact of using such a model on the accuracy and diversity of the recommender system.

5. Discuss how you would control the relative strength of collaborative and content-based portions in Exercise 4 with a single weight parameter. How would you determine the optimal value of the weight parameter in a data-driven way?

Chapter 7

Evaluating Recommender Systems

"*True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.*"– Winston Churchill

7.1 Introduction

The evaluation of collaborative filtering shares a number of similarities with that of classification. This similarity is due to the fact that collaborative filtering can be viewed as a generalization of the classification and regression modeling problem (cf. section 1.3.1.3 of Chapter 1). Nevertheless, there are many aspects to the evaluation process that are unique to collaborative filtering applications. The evaluation of content-based methods is even more similar to that of classification and regression modeling, because content-based methods often use text classification methods under the covers. This chapter will introduce various mechanisms for evaluating various recommendation algorithms and also relate these techniques to the analogous methods used in classification and regression modeling.

A proper design of the evaluation system is crucial in order to obtain an understanding of the effectiveness of various recommendation algorithms. As we will see later in this chapter, the evaluation of recommender systems is often multifaceted, and a single criterion cannot capture many of the goals of the designer. An incorrect design of the experimental evaluation can lead to either gross underestimation or overestimation of the true accuracy of a particular algorithm or model.

Recommender systems can be evaluated using either *online* methods or *offline* methods. In an online system, the user reactions are measured with respect to the presented recommendations. Therefore, user participation is essential in online systems. For example, in an online evaluation of a news recommender system, one might measure the *conversion rate* of users clicking on articles that were recommended. Such testing methods are referred to as *A/B testing*, and they measure the direct impact of the recommender system

C.C. Aggarwal, *Recommender Systems: The Textbook*,
DOI 10.1007/978-3-319-29659-3_7

on the end user. At the end of the day, increasing the conversion rate on profitable items is the most important goal of a recommender system, and it can provide a true measure of the effectiveness of the system. However, since online evaluations require active user participation, it is often not feasible to use them in benchmarking and research. There are usually significant challenges in gaining access to user conversion data from systems with large-scale user participation. Even if such access is gained, it is usually specific to a single large-scale system. On the other hand, one often desires to use data sets of different types, and from multiple domains. Testing over multiple data sets is particularly important for assuring greater generalization power of the recommender system so that one can be assured that the algorithm works under a variety of settings. In such cases, offline evaluations with historical data sets are used. Offline methods are, by far, the most common methods for evaluating recommender systems from a research and practice perspective. Therefore, most of this chapter will focus on offline methods, although some discussion of online methods is also included for completeness.

When working with offline methods, accuracy measures can often provide an incomplete picture of the true conversion rate of a recommender system. Several other secondary measures also play a role. Therefore, it is important to design the evaluation system carefully so that the measured metrics truly reflect the effectiveness of the system from the user perspective. In particular, the following issues are important from the perspective of designing evaluation methods for recommender systems:

1. *Evaluation goals:* While it is tempting to use accuracy metrics for evaluating recommender systems, such an approach can often provide an incomplete picture of the user experience. Although accuracy metrics are arguably the most important components of the evaluation, many secondary goals such as novelty, trust, coverage, and serendipity are important to the user experience. This is because these metrics have important short- and long-term impacts on the conversion rates. Nevertheless, the actual quantification of some of these factors is often quite subjective, and there are often no hard measures to provide a numerical metric.

2. *Experimental design issues:* Even when accuracy is used as the metric, it is crucial to design the experiments so that the accuracy is not overestimated or underestimated. For example, if the same set of specified ratings is used both for model construction and for accuracy evaluation, then the accuracy will be grossly overestimated. In this context, careful experimental design is important.

3. *Accuracy metrics:* In spite of the importance of other secondary measures, accuracy metrics continue to be the single most important component in the evaluation. Recommender systems can be evaluated either in terms of the prediction accuracy of a rating or the accuracy of ranking the items. Therefore, a number of common metrics such as the *mean absolute error* and *mean squared error* are used frequently. The evaluation of rankings can be performed with the use of various methods, such as utility-based computations, rank-correlation coefficients, and the *receiver operating characteristic* curve.

In this chapter, we will first begin by discussing the general goals of evaluating recommender systems beyond the most basic criterion of accuracy. Examples of such goals include diversity and novelty. The main challenge with *quantifying* such goals is that they are often subjective goals based on user experience. From a quantification perspective, accuracy is a concrete goal that is relatively easy to measure and is therefore used more frequently for bench-marking and testing. A few quantification methods do exist for evaluating the secondary goals such as diversity and novelty. Although the majority of this chapter will focus on accuracy metrics, various quantification measures for the secondary goals will also be discussed.

This chapter is organized as follows. An overview of the different types of evaluation systems is provided in section 7.2. Section 7.3 studies the general goals of evaluating recommender systems. The appropriate design of accuracy testing methods is discussed in section 7.4. Accuracy metrics for recommender systems are discussed in section 7.5. The limitations of evaluation measures are discussed in 7.6. A summary is given in section 7.7.

7.2 Evaluation Paradigms

There are three primary types of evaluation of recommender systems, corresponding to user studies, online evaluations, and offline evaluations with historical data sets. The first two types involve users, although they are conducted in slightly different ways. The main differences between the first two settings lie in how the users are recruited for the studies. Although online evaluations provide useful insights about the true effects of a recommendation algorithm, there are often significant practical impediments in their deployment. In the following, an overview of these different types of evaluation is provided.

7.2.1 User Studies

In user studies, test subjects are actively recruited, and they are asked to interact with the recommender system to perform specific tasks. Feedback can be collected from the user before and after the interaction, and the system also collects information about their interaction with the recommender system. These data are then used to make inferences about the likes or dislikes of the user. For example, users could be asked to interact with the recommendations at a product site and give their feedback about the quality of the recommendations. Such an approach could then be used to judge the effectiveness of the underlying algorithms. Alternatively, users could be asked to listen to several songs, and then provide their feedback on these songs in the form of ratings.

An important advantage of user studies is that they allow for the collection of information about the user interaction with the system. Various scenarios can be tested about the effect of changing the recommender system on the user interaction, such as the effect of changing a particular algorithm or user-interface. On the other hand, the active awareness of the user about the testing of the recommender system can often bias her choices and actions. It is also difficult and expensive to recruit large cohorts of users for evaluation purposes. In many cases, the recruited users are not representative of the general population because the recruitment process is itself a bias-centric filter, which cannot be fully controlled. Not all users would be willing to participate in such a study, and those who do agree might have unrepresentative interests with respect to the remaining population. For example, in the case of the example of rating songs, the (voluntary) participants are likely to be music enthusiasts. Furthermore, the fact that users are actively aware of their recruitment for a particular study is likely to affect their responses. Therefore, the results from user evaluations cannot be fully trusted.

7.2.2 Online Evaluation

Online evaluations also leverage user studies except that the users are often real users in a fully deployed or commercial system. This approach is sometimes less susceptible to bias from the recruitment process, because the users are often directly using the system in the natural course of affairs. Such systems can often be used to evaluate the comparative performance of various algorithms [305]. Typically, users can be sampled randomly, and the various algorithms can be tested with each sample of users. A typical example of a metric, which is used to measure the effectiveness of the recommender system on the users, is the *conversion rate*. The conversion rate measures the frequency with which a user selects a recommended item. For example, in a news recommender system, one might compute the fraction of times that a user selects a recommended article. If desired, expected costs or profits can be added to the items to make the measurement sensitive to the importance of the item. These methods are also referred to as *A/B testing*, and they measure the direct impact of the recommender system on the end user. The basic idea in these methods is to compare two algorithms as follows:

1. Segment the users into two groups A and B.

2. Use one algorithm for group A and another algorithm for group B for a period of time, while keeping all other conditions (e.g., selection process of users) across the two groups as similar as possible.

3. At the end of the process, compare the conversion rate (or other payoff metric) of the two groups.

This approach is very similar to what is used for clinical trials in medicine. Such an approach is the most accurate one for testing the long-term performance of the system directly in terms of goals such as profit. These methods can also be leveraged for the user studies discussed in the previous section.

One observation is that it is not necessary to strictly segment the users into groups in cases where the payoff of each interaction between the user and the recommender can be measured separately. In such cases, the same user can be shown one of the algorithms at random, and the payoff from that specific interaction can be measured. Such methods of evaluating recommender systems have also been generalized to the development of more effective recommendation algorithms. The resulting algorithms are referred to as *multi-arm bandit algorithms*. The basic idea is similar to that of a gambler (recommender system) who is faced with a choice of selecting one of a set of slot machines (recommendation algorithms) at the casino. The gambler suspects that one of these machines has a better payoff (conversion rate) than others. Therefore, the gambler tries a slot machine at random 10% of the time in order to *explore* the relative payoffs of the machines. The gambler greedily selects the best paying slot machine the remaining 90% of the time in order to *exploit* the knowledge learned in the exploratory trials. The process of exploration and exploitation is fully interleaved in a random way. Furthermore, the gambler may choose to give greater weight to recent results as compared to older results for evaluation. This general approach is related to the notion of *reinforcement learning*, which can often be paired with online systems. Although reinforcement learning has been studied extensively in the classification and regression modeling literature [579], the corresponding work in the recommendation domain is rather limited [389, 390, 585]. A significant research opportunity exists for the further development of such algorithms.

The main disadvantage is that such systems cannot be realistically deployed unless a large number of users are already enrolled. Therefore, it is hard to use this method during the start up phase. Furthermore, such systems are usually not openly accessible, and they are only accessible to the owner of the specific commercial system at hand. Therefore, such tests can be performed only by the commercial entity, and for the limited number of scenarios handled by their system. This means that the tests are often not generalizable to system-independent benchmarking by scientists and practitioners. In many cases, it is desirable to test the robustness of a recommendation algorithm by stress-testing it under a variety of settings and data domains. By using multiple settings, one can obtain an idea of the generalizability of the system. Unfortunately, online methods are not designed for addressing such needs. A part of the problem is that one cannot fully control the actions of the test users in the evaluation process.

7.2.3 Offline Evaluation with Historical Data Sets

In offline testing, historical data, such as ratings, are used. In some cases, temporal information may also be associated with the ratings, such as the time-stamp at which each user has rated the item. A well known example of a historical data set is the Netflix Prize data set [311]. This data set was originally released in the context of an online contest, and has since been used as a standardized benchmark for testing many algorithms. The main advantage of the use of historical data sets is that they do not require access to a large user base. Once a data set has been collected, it can be used as a standardized benchmark to compare various algorithms across a variety of settings. Furthermore, multiple data sets from various domains (e.g., music, movies, news) can be used to test the generalizability of the recommender system.

Offline methods are among the most popular techniques for testing recommendation algorithms, because standardized frameworks and evaluation measures have been developed for such cases. Therefore, much of this chapter will be devoted to the study of offline evaluation. The main disadvantage of offline evaluations is that they do not measure the actual propensity of the user to react to the recommender system in the future. For example, the data might evolve over time, and the current predictions may not reflect the most appropriate predictions for the future. Furthermore, measures such as accuracy do not capture important characteristics of recommendations, such as *serendipity* and *novelty*. Such recommendations have important long-term effects on the conversion rate of the recommendations. Nevertheless, in spite of these disadvantages, offline methods continue to be the most widely accepted techniques for recommender system evaluation. This is because of the statistically robust and easily understandable quantifications available through such testing methods.

7.3 General Goals of Evaluation Design

In this section, we will study some of the general goals in evaluating recommender systems. Aside from the well known goal of accuracy, other general goals include factors such as diversity, serendipity, novelty, robustness, and scalability. Some of these goals can be concretely quantified, whereas others are subjective goals based on user experience. In such cases, the only way of measuring such goals is through user surveys. In this section, we will study these different goals.

7.3.1 Accuracy

Accuracy is one of the most fundamental measures through which recommender systems are evaluated. In this section, we provide a brief introduction to this measure. A detailed discussion is provided in section 7.5 of this chapter. In the most general case, ratings are numeric quantities that need to be estimated. Therefore, the accuracy metrics are often similar to those used in regression modeling. Let R be the ratings matrix in which r_{uj} is the known rating of user u for item j. Consider the case where a recommendation algorithm estimates this rating as $\hat{r}_{uj}$. Then, the *entry-specific* error of the estimation is given by the quantity $e_{uj} = \hat{r}_{uj} - r_{uj}$. The overall error is computed by averaging the entry-specific errors either in terms of absolute values or in terms of squared values. Furthermore, many systems do not predict ratings; rather they only output rankings of top-k recommended items. This is particularly common in implicit feedback data sets. Different methods are used to evaluate the accuracy of ratings predictions and the accuracy of rankings.

As the various methods for computing accuracy are discussed in detail in section 7.5, they are not discussed in detail here. The goal of this short section is to briefly introduce a few measures to ensure continuity in further discussion. The main components of accuracy evaluation are as follows:

1. *Designing the accuracy evaluation:* All the observed entries of a ratings matrix cannot be used both for training the model and for accuracy evaluation. Doing so would grossly overestimate the accuracy because of overfitting. It is important to use only a different set of entries for evaluation than was used for training. If S is the observed entries in the ratings matrix, then a small subset $E \subset S$ is used for evaluation, and the set $S - E$ is used for training. This issue is identical to that encountered in the evaluation of classification algorithms. After all, as discussed in earlier chapters, collaborative filtering is a direct generalization of the classification and regression modeling problem. Therefore, the standard methods that are used in classification and regression modeling, such as hold-out and cross-validation, are also used in the evaluation of recommendation algorithms. These issues will be discussed in greater detail in section 7.4.

2. *Accuracy metrics:* Accuracy metrics are used to evaluate either the prediction accuracy of estimating the ratings of specific user-item combinations or the accuracy of the top-k ranking predicted by a recommender system. Typically, the ratings of a set E of entries in the ratings matrix are hidden, and the accuracy is evaluated over these hidden entries. Different classes of methods are used for the two cases:

   - *Accuracy of estimating ratings:* As discussed above, the entry-specific error is given by $e_{uj} = \hat{r}_{uj} - r_{uj}$ for user u and item j. This error can be leveraged in various ways to compute the overall error over the set E of entries in the ratings matrix on which the evaluation is performed. An example is the *mean squared error*, which is denoted by *MSE*:

     $$MSE = \frac{\sum_{(u,j)\in E} e_{uj}^2}{|E|} \tag{7.1}$$

     The square-root of the aforementioned quantity is referred to as the *root mean squared error*, or *RMSE*.

     $$RMSE = \sqrt{\frac{\sum_{(u,j)\in E} e_{uj}^2}{|E|}} \tag{7.2}$$

     Most of these measures are borrowed from the literature on regression modeling. Other important ways of measuring the error, such as the mean absolute error, are discussed in section 7.5.

- *Accuracy of estimating rankings:* Many recommender systems do not directly estimate ratings; instead, they provide estimates of the underlying ranks. Depending on the nature of the ground-truth, one can use rank-correlation measures, utility-based measures, or the receiver operating characteristic. The latter two methods are designed for unary (implicit feedback) data sets. These methods are discussed in detail in section 7.5.

Some measures of accuracy are also designed to maximize the profit for the merchant because all items are not equally important from the perspective of the recommendation process. These metrics incorporate item-specific costs into the computation. The main problem with accuracy metrics is that they often do not measure the true effectiveness of a recommender system in real settings. For example, an obvious recommendation might be accurate, but a user might have eventually bought that item anyway. Therefore, such a recommendation might have little usefulness in terms of improving the conversion rate of the system. A discussion of the challenges associated with the use of accuracy metrics may be found in [418].

7.3.2 Coverage

Even when a recommender system is highly accurate, it may often not be able to ever recommend a certain proportion of the items, or it may not be able to ever recommend to a certain proportion of the users. This measure is referred to as *coverage*. This limitation of recommender systems is an artifact of the fact that ratings matrices are sparse. For example, in a rating matrix contains a single entry for each row and each column, then no meaningful recommendations can be made by almost *any* algorithm. Nevertheless, different recommender systems have different levels of propensity in providing coverage. In practical settings, the systems often have 100% coverage because of the use of defaults for ratings that are not possible to predict. An example of such a default would be to report the average of all the ratings of a user for an item when the rating for a specific user-item combination cannot be predicted. Therefore, the trade-off between accuracy and coverage always needs to be incorporated into the evaluation process. There are two types of coverage, which are referred to as *user-space coverage* and *item-space coverage*, respectively.

User-space coverage measures the fraction of users for which at least k ratings may be predicted. The value of k should be set to the expected size of the recommendation list. When fewer than k ratings can be predicted for a user, it is no longer possible to present a meaningful recommendation list of size k to the user. Such a situation could occur when a user has specified very few ratings in common with other users. Consider a user-based neighborhood algorithm. It is difficult to robustly compute the peers of that user, because of very few mutually specified ratings with other users. Therefore, it is often difficult to make sufficient recommendations for that user. For very high levels of sparsity, it is possible that no algorithm may be able to predict even one rating for that user. However, different algorithms may have different levels of coverage, and the coverage of a user can be estimated by running each algorithm and determining the number of items for which a prediction is made. A tricky aspect of user-space coverage is that any algorithm can provide full coverage by simply predicting random ratings for user-item combinations, whose ratings it cannot reliably predict. Therefore, user-space coverage should always be evaluated in terms of the trade-off between accuracy and coverage. For example, in a neighborhood-based recommender increasing the size of the neighborhood provides a curve showing the trade-off between coverage and accuracy.

An alternative definition of user-space coverage is in terms of the minimum amount of profile that must be built for a user before it is possible to make recommendations for that user. For a particular algorithm, it is possible to estimate through experiments the minimum number of observed ratings of any user for which a recommendation could be made. However, it is often difficult to evaluate this quantity because the metric is sensitive to the identity of the items for which the user specifies ratings.

The notion of *item-space coverage* is analogous to that of user-space coverage. Item-space coverage measures the fraction of items for which the ratings of at least k users can be predicted. In practice, however, this notion is rarely used, because recommender systems generally provide recommendation lists for users, and they are only rarely used for generating recommended users for items.

A different form of item-space coverage evaluation is defined by the notion of *catalog coverage*, which is specifically suited to recommendation *lists*. Note that the aforementioned definition was tailored to the prediction of the values of ratings. Imagine a scenario where every entry in the ratings matrix can be predicted by an algorithm, but the same set of top-k items is always recommended to every user. Therefore, even though the aforementioned definition of item-space coverage would suggest good performance, the actual coverage across all users is very limited. In other words, the recommendations are not diverse across users, and the catalog of items is not fully covered. Let T_u represent the list of top-k items recommended to user $u \in \{1 \dots m\}$. The catalog coverage CC is defined as the fraction of items that are recommended to at least one user.

$$CC = \frac{| \cup_{u=1}^{m} T_u|}{n} \tag{7.3}$$

Here, the notation n represents the number of items. It is easy to estimate this fraction through the use of experiments.

7.3.3 Confidence and Trust

The estimation of ratings is an inexact process that can vary significantly with the specific training data at hand. Furthermore, the algorithmic methodology might also have a significant impact on the predicted ratings. This always leads to uncertainty in the user about the accuracy of the predictions. Many recommender systems may report ratings together with confidence estimates. For example, a confidence interval on the range of predicted ratings may be provided. In general, recommender systems that can accurately recommend smaller confidence intervals are more desirable because they bolster the user's trust in the system. For two algorithms that use the same method for reporting confidence, it is possible to measure how well the predicted error matches these confidence intervals. For example, if two recommender systems provide 95% confidence intervals for each rating, one can measure the absolute width of the intervals reported by the two algorithms. The algorithm with the smaller confidence interval width will win as long as both algorithms are correct (i.e., within the specified intervals) at least 95% of the time on the hidden ratings. If one of the algorithms falls below the required 95% accuracy, then it automatically loses. Unfortunately, if one system uses 95% confidence intervals and another uses 99% confidence intervals, it is not possible to meaningfully compare them. Therefore, it is possible to use such systems only by setting the same level of confidence in both cases.

While confidence measures the system's faith in the recommendation, trust measures the user's faith in the evaluation. The notion of social trust is discussed in more detail in Chapter 11. Broadly speaking, trust measures the level of faith that the *user* has in the reported ratings. Even if the predicted ratings are accurate, they are often not useful if the user fails to trust the provided ratings. Trust is closely related to, but not quite the same as, accuracy. For example, when explanations are provided by the recommender system, the user is more likely to trust the system, especially if the explanations are logical.

Trust often does not serve the same goals as the usefulness (utility) of a recommendation. For example, if a recommender system suggests a few items already liked and known by the user, it can be argued that there is little utility provided to the user from such a recommendation. On the other hand, such items can increase the trust of the user in the system. This goal is directly in contradiction to other goals such as novelty in which recommendations already known by the user are undesirable. It is common for the various goals in recommender systems to trade-off against one another. The simplest way to measure trust is to conduct user surveys during the experiments in which the users are explicitly queried about their trust in the results. Such experiments are also referred to as *online experiments*. Numerous online methods for trust evaluation are discussed in [171, 175, 248, 486]. Generally, it is hard to measure trust through offline experiments.

7.3.4 Novelty

The novelty of a recommender system evaluates the likelihood of a recommender system to give recommendations to the user that they are not aware of, or that they have not seen before. A discussion of the notion of novelty is provided in [308]. Unseen recommendations often increase the ability of the user to discover important insights into their likes and dislikes that they did not know previously. This is more important than discovering items that they were already aware of but they have not rated. In many types of recommender systems, such as content-based methods, the recommendations tend to be somewhat obvious because of the propensity of the system to recommend expected items. While a small number of such recommendations can improve the trust of the end user in the underlying system, they are not always useful in terms of improving conversion rates. The most natural way of measuring novelty is through online experimentation in which users are explicitly asked whether they were aware of an item previously.

As discussed in the introduction, online experimentation is not always feasible because of the lack of access to a system supporting a large base of online users. Fortunately, it is possible to approximately estimate novelty with offline methods, as long as time stamps are available with the ratings. The basic idea is that novel systems are better at recommending items that are more likely to be selected by the user in the *future*, rather than at the present time. Therefore, all ratings that were created after a certain point in time t_0 are removed from the training data. Furthermore, some of the ratings occurring before t_0 are also removed. The system is then trained with these ratings removed. These removed items are then used for scoring purposes. For each item rated before time t_0 and correctly recommended, the novelty evaluation score is penalized. On the other hand, for each item rated after time t_0 and correctly recommended, the novelty evaluation score is rewarded. Therefore, this evaluation measures a type of *differential* accuracy between future and past predictions. In some measures of novelty, it is assumed that popular items are less likely to be novel, and less credit is given for recommending popular items.

7.3.5 Serendipity

The word “serendipity” literally means “lucky discovery.” Therefore, serendipity is a measure of the level of surprise in successful recommendations. In other words, recommendations need to be *unexpected*. In contrast, novelty only requires that the user was not *aware* of the recommendation earlier. Serendipity is a stronger condition than novelty. All serendipitious recommendations are novel, but the converse is not always true. Consider the case where a particular user frequently eats at Indian restaurants. The recommendation of a new Pakistani restaurant to that user might be novel if that user has not eaten at that restaurant earlier. However, such a recommendation is not serendipitious, because it is well known that Indian and Pakistani food are almost identical. On the other hand, if the recommender system suggests a new Ethiopian restaurant to the user, then such a recommendation is serendipitious because it is less obvious. Therefore, one way of viewing serendipity is as a departure from "obviousness."

There are several ways of measuring serendipity in recommender systems. This notion also appears in the context of information retrieval applications [670]. The work in [214] proposed both online and offline methods for evaluating serendipity:

1. *Online methods:* The recommender system collects user feedback both on the usefulness of a recommendation and its obviousness. The fraction of recommendations that are both useful and non-obvious, is used as a measure of the serendipity.

2. *Offline methods:* One can also use a primitive recommender to generate the information about the obviousness of a recommendation in an automated way. The primitive recommender is typically selected as a content-based recommender, which has a high propensity for recommending obvious items. Then, the fraction of the recommended items in the top-k lists that are correct (i.e., high values of hidden ratings), and are also not recommended by the primitive recommender are determined. This fraction provides a measure of the serendipity.

It is noteworthy that it is not sufficient to measure the fraction of non-obvious items, because a system might recommend unrelated items. Therefore, the usefulness of the items is always incorporated in the measurement of serendipity. Serendipity has important long-term effects on improving the conversion rate of a recommender system, even when it is opposed to the immediate goal of maximizing accuracy. A number of metrics for serendipity evaluation are discussed in [214, 450].

7.3.6 Diversity

The notion of diversity implies that the set of proposed recommendations *within a single recommended list* should be as diverse as possible. For example, consider the case where three movies are recommended to a user in the list of top-3 items. If all three movies are of a particular genre and contain similar actors, then there is little diversity in the recommendations. If the user dislikes the top choice, then there is a good chance that she might dislike all of them. Presenting different types of movies can often increase the chance that the user might select one of them. Note that the diversity is always measured over a *set* of recommendations, and it is closely related to novelty and serendipity. Ensuring greater diversity can often increase the novelty and serendipity of the recommendations. Furthermore, greater diversity of recommendations can also increase the sales diversity and catalog coverage of the system.

Diversity can be measured in terms of the content-centric similarity between pairs of items. The vector-space representation of each item description is used for the similarity computation. For example, if a set of k items are recommended to the user, then the pairwise similarity is computed between every pair of items in the list. The average similarity between all pairs can be reported as the diversity. Lower values of the average similarity indicate greater diversity. Diversity can often provide very different results from those of accuracy metrics. A discussion of the connection of diversity and similarity is provided in [560].

7.3.7 Robustness and Stability

A recommender system is stable and robust when the recommendations are not significantly affected in the presence of attacks such as fake ratings or when the patterns in the data evolve significantly over time. In general, significant profit-driven motivations exist for some users to enter fake ratings [158, 329, 393, 444]. For example, the author or publisher of a book might enter fake positive ratings about a book at Amazon.com, or they might enter fake negative ratings about the books of a rival. Attack models for recommender systems are discussed in Chapter 12. The evaluation of such models is also studied in the same chapter. The corresponding measures can be used to estimate the robustness and stability of such systems against attacks.

7.3.8 Scalability

In recent years, it has become increasingly easy to collect large numbers of ratings and implicit feedback information from various users. In such cases, the sizes of the data sets continue to increase over time. As a result, it has become increasingly essential to design recommender systems that can perform effectively and efficiently in the presence of large amounts of data [527, 528, 587]. A variety of measures are used for determining the scalability of a system:

1. *Training time:* Most recommender systems require a training phase, which is separate from the testing phase. For example, a neighborhood-based collaborative filtering algorithm might require pre-computation of the peer group of a user, and a matrix factorization system requires the determination of the latent factors. The overall time required to train a model is used as one of the measures. In most cases, the training is done offline. Therefore, as long as the training time is of the order of a few hours, it is quite acceptable in most real settings.

2. *Prediction time:* Once a model has been trained, it is used to determine the top recommendations for a particular customer. It is crucial for the prediction time to be low, because it determines the latency with which the user receives the responses.

3. *Memory requirements:* When the ratings matrices are large, it is sometimes a challenge to hold the entire matrix in the main memory. In such cases, it is essential to design the algorithm to minimize memory requirements. When the memory requirements become very high, it is difficult to use the systems in large-scale and practical settings.

The importance of scalability has become particularly great in recent years because of the increasing importance of the "big-data" paradigm.

7.4 Design Issues in Offline Recommender Evaluation

In this section, we will discuss the issue of recommender evaluation design. The discussions in this section and the next pertain to *accuracy* evaluation of offline and historical data sets. It is crucial to design recommender systems in such a way that the accuracy is not grossly overestimated or underestimated. For example, one cannot use the same set of specified ratings for both training and evaluation. Doing so would grossly overestimate the accuracy of the underlying algorithm. Therefore, only a part of the data is used for training, and the remainder is often used for testing. The ratings matrix is typically sampled in an *entry-wise fashion.* In other words, a subset of the entries are used for training, and the remaining entries are used for accuracy evaluation. Note that this approach is similar to that used for testing classification and regression modeling algorithms. The main difference is that classification and regression modeling methods sample *rows* of the labeled data, rather than sampling the *entries.* This difference is because the unspecified entries are always restricted to the class variable in classification, whereas any entry of the ratings matrix can be unspecified. The design of recommender evaluation systems is very similar to that of classifier evaluation systems because of the similarity between the recommendation and classification problems.

A common mistake made by analysts in the benchmarking of recommender systems is to use the same data for parameter tuning and for testing. Such an approach grossly overestimates the accuracy because parameter tuning is a part of training, and the use of test data in the training process leads to overfitting. To guard against this possibility, the data are often divided into three parts:

1. *Training data:* This part of the data is used to build the training model. For example, in a latent factor model, this part of the data is used to create the latent factors from the ratings matrix. One might even use these data to create multiple models in order to eventually select the model that works best for the data set at hand.

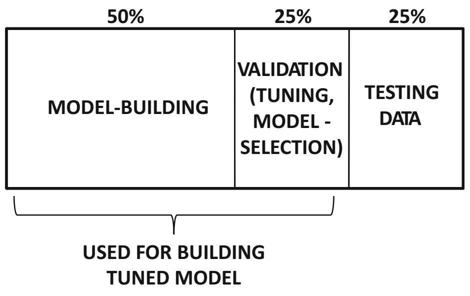
2. *Validation data:* This part of the data is used for model selection and parameter tuning. For example, the regularization parameters in a latent factor model may be determined by testing the accuracy over the validation data. In the event that multiple models have been built from the training data, the validation data are used to determine the accuracy of each model and select the best one.

3. *Testing data:* This part of the data is used to test the accuracy of the final (tuned) model. It is important that the testing data are not even looked at during the process of parameter tuning and model selection to prevent overfitting. The testing data are *used only once at the very end of the process.* Furthermore, if the analyst uses the results on the test data to adjust the model in some way, then the results will be contaminated with knowledge from the testing data.

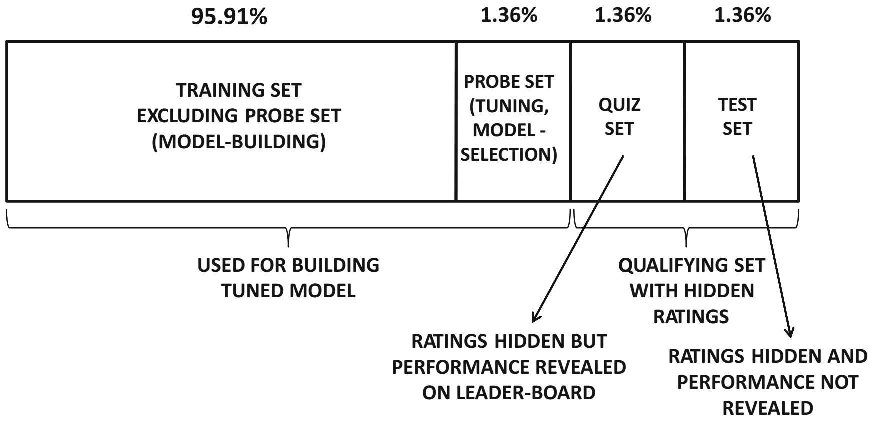
An example of a division of the ratings matrix into training, validation, and testing data is illustrated in Figure 7.1(a). Note that the validation data may also be considered a part of the training data because they are used to create the final tuned model. The division of the ratings matrix into the ratios 2:1:1 is particularly common. In other words, half the specified ratings are used for model-building, and a quarter may be used for each of model-selection and testing, respectively. However, when the sizes of the ratings matrices are large, it is possible to use much smaller proportions for validation and testing. This was the case for the Netflix Prize data set.

7.4.1 Case Study of the Netflix Prize Data Set

A particularly instructive example of a well-known data set used in collaborative filtering is the Netflix Prize data set, because it demonstrates the extraordinary lengths to which Netflix went to prevent overfitting on the test set from the contest participants. In the Netflix data


50%
25%
25%
MODEL-BUILDING
VALIDATION (TUNING, MODEL - SELECTION)
TESTING DATA
USED FOR BUILDING TUNED MODEL

(a) Proportional division of ratings


95.91%
1.36%
1.36%
1.36%
TRAINING SET EXCLUDING PROBE SET (MODEL-BUILDING)
PROBE SET (TUNING, MODEL - SELECTION)
QUIZ SET
TEST SET
USED FOR BUILDING TUNED MODEL
QUALIFYING SET WITH HIDDEN RATINGS
RATINGS HIDDEN BUT PERFORMANCE REVEALED ON LEADER-BOARD
RATINGS HIDDEN AND PERFORMANCE NOT REVEALED

(b) Division in Netflix Prize data set (not drawn to scale)

Figure 7.1: Partitioning a ratings matrix for evaluation design

set, the largest portion of the data set contained 95.91% of the ratings. This portion of the data set was typically used by the contest participants for model-building. Another 1.36% of the data set was revealed to the participants as a *probe set*. Therefore, the model-building portion of the data and the probe data together contained $95.91 + 1.36 = 97.27\%$ of the data. The probe set was typically used by contests for various forms of parameter tuning and model selection, and therefore it served a very similar purpose as a validation set. However, different contestants used the probe set in various ways, especially since the ratings in the probe set were more recent, and the statistical distribution of the ratings in the training and probe sets were slightly different. For the case of ensemble methods [554], the probe set was often used to learn the weights of various ensemble components. The combined data set with revealed ratings (including the probe set) corresponds to the full training data, because it was used to build the final tuned model. An important peculiarity of the training data was that the distributions of the probe set and the model-building portion of the training set were not exactly identical, although the probe set reflected the statistical characteristics of the *qualifying set* with hidden ratings. The reason for this difference was that most of the ratings data were often quite old and they did not reflect the true distribution of the more recent or future ratings. The probe and qualifying sets were based on more recent ratings, compared to the 95.91% of the ratings in the first part of the training data.

The ratings of the remaining 2.7% of the data were hidden, and only triplets of the form $\langle User, Movie, GradeDate \rangle$ were supplied without actual ratings. The main difference from a test set was that participants could submit their performance on the qualifying set to Netflix, and the performance on half the qualifying data, known as the *quiz set*, was revealed to the participants on a *leader-board*. Although revealing the performance on the quiz set to the participants was important in order to give them an idea of the quality of their results, the problem with doing so was that participants could use the knowledge of the performance of their algorithm on the leader-board to over-train their algorithm on the quiz set with repeated submissions. Clearly, doing so results in contamination of the results from knowledge of the performance on the quiz set, even when ratings are hidden. Therefore, the part of the qualifying set that was *not* in the quiz set was used as the test set, and the results on *only* this part of the qualifying set were used to determine the final performance for the purpose of prize determination. The performance on the quiz set had no bearing on the final contest, except to give the participants a continuous idea of their performance during the contest period. Furthermore, the participants were not informed about which part of the qualifying set was the quiz set. This arrangement ensured that a truly out-of-sample data set was used to determine the final winners of the contest.

The overall division of the Netflix data set is shown in Figure 7.1(b). The only difference from the division in Figure 7.1(a) is the presence of an additional quiz set. It is, in fact, possible to remove the quiz set entirely without affecting the Netflix contest in any significant way, except that participants would no longer be able to obtain an idea of the quality of their submissions. Indeed, the Netflix Prize evaluation design is an excellent example of the importance of not using any knowledge of the performance on the test set at any stage of the training process *until the very end*. Benchmarking in research and practice often fails to meet these standards in one form or the other.

7.4.2 Segmenting the Ratings for Training and Testing

In practice, real data sets are not pre-partitioned into training, validation, and test data sets. Therefore, it is important to be able to divide the entries of a ratings matrix into these portions automatically. Most of the available division methods, such as *hold-out* and *cross-validation*, are used to divide[1] the data set into *two* portions instead of *three*. However, it is possible to obtain three portions as follows. By first dividing the rating entries into training and test portions, and then further segmenting the validation portion from the training data, it is possible to obtain the required three segments. Therefore, in the following, we will discuss the segmentation of the ratings matrix into training and testing portions of the entries using methods such as hold-out and cross-validation. However, these methods are also used for dividing the training data into the model-building and validation portions. This hierarchical division is illustrated in Figure 7.2. In the following, we will consistently use the terminology of the first level of division in Figure 7.2 into "training" and "testing" data, even though the same approach can also be used for the second level division into model building and validation portions. This consistency in terminology is followed to avoid confusion.

7.4.2.1 Hold-Out

In the hold-out method, a fraction of the entries in the ratings matrix are hidden, and the remaining entries are used to build the training model. The accuracy of predicting the hidden entries is then reported as the overall accuracy. Such an approach ensures that the reported accuracy is not a result of overfitting to the specific data set, because the entries

[1]The actual design in methods such as cross-validation is slightly more complex because the data are segmented in multiple ways, even though they are always divided into two parts during a particular execution phase of training.

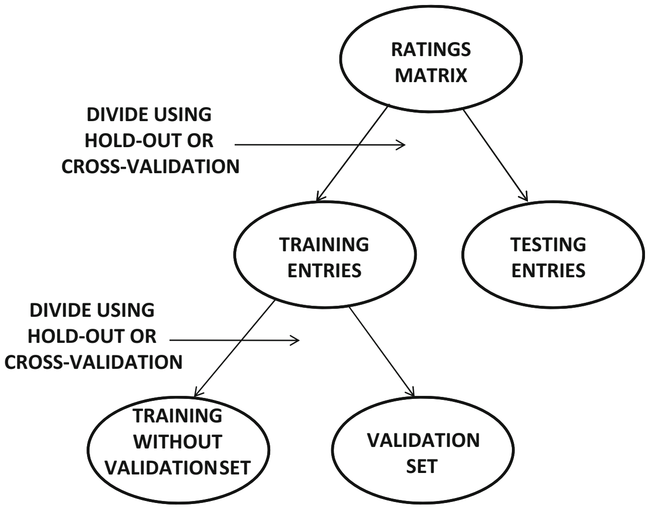

RATINGS MATRIX
DIVIDE USING HOLD-OUT OR CROSS-VALIDATION
TRAINING ENTRIES
TESTING ENTRIES
DIVIDE USING HOLD-OUT OR CROSS-VALIDATION
TRAINING WITHOUT VALIDATIONSET
VALIDATION SET

Figure 7.2: Hierarchical division of rated entries into training, validation, and testing portions

used for evaluation are hidden during training. Such an approach, however, underestimates the true accuracy. First, all entries are not used in training, and therefore the full power of the data is not used. Second, consider the case where the held-out entries have a higher average rating than the full ratings matrix. This means that the held-in entries have a lower average rating than the ratings matrix, and also the held-out entries. This will lead to a pessimistic bias in the evaluation.

7.4.2.2 Cross-Validation

In the cross-validation method, the ratings entries are divided into q equal sets. Therefore, if S is the set of specified entries in the ratings matrix R, then the size of each set, in terms of the number of entries, is $|S|/q$. One of the q segments is used for testing, and the remaining $(q-1)$ segments are used for training. In other words, a total of $|S|/q$ entries are hidden during *each* such training process, and the accuracy is then evaluated over these entries. This process is repeated q times by using each of the q segments as the test set. The average accuracy over the q different test sets is reported. Note that this approach can closely estimate the true accuracy when the value of q is large. A special case is one where q is chosen to be equal to the number of specified entries in the ratings matrix. Therefore, $|S|-1$ rating entries are used for training, and the one entry is used for testing. This approach is referred to as *leave-one-out cross-validation*. Although such an approach can closely approximate the accuracy, it is usually too expensive to train the model $|S|$ times. In practice, the value of q is fixed to a number such as 10. Nevertheless, leave-one-out cross-validation is not very difficult to implement for the specific case of neighborhood-based collaborative filtering algorithms.

7.4.3 Comparison with Classification Design

The evaluation design in collaborative filtering is very similar to that in classification. This is not a coincidence. Collaborative filtering is a generalization of the classification problem, in which any missing entry can be predicted rather than simply a particular variable, which is designated as the dependent variable. The main difference from classification is that the data are segmented on a row-wise basis (between training and test *rows*) in classification, whereas the data are segmented on an entry-wise basis (between training and test *entries*) in collaborative filtering. This difference closely mirrors the nature of the relationship between the classification and the collaborative filtering problems. Discussions of evaluation designs in the context of the classification problem can be found in [18, 22].

One difference from classification design is that the performance on hidden entries often does not reflect the true performance of the system in real settings. This is because the hidden ratings are not chosen at random from the matrix. Rather, the hidden ratings are typically items that the user has chosen to consume. Therefore, such entries are likely to have higher values of the ratings as compared to truly missing values. This is a problem of *sample selection bias*. Although this problem could also arise in classification, it is far more pervasive in collaborative filtering applications. A brief discussion of this issue is provided in section 7.6.

7.5 Accuracy Metrics in Offline Evaluation

Offline evaluation can be performed by measuring the accuracy of predicting rating values (e.g., with *RMSE*) or by measuring the accuracy of ranking the recommended items. The logic for the latter set of measures is that recommender systems often provide ranked lists of items without explicitly predicting ratings. Ranking-based measures often focus on the accuracy of only the ranks of the top-k items rather than all the items. This is particularly true in the case of implicit feedback data sets. Even in the case of explicit ratings, the ranking-based evaluations provide a more realistic perspective of the true usefulness of the recommender system because the user only views the top-k items rather than all the items. However, for bench-marking, the accuracy of ratings predictions is generally preferred because of its simplicity. In the Netflix Prize competition, the *RMSE* measure was used for final evaluation. In the following, both forms of accuracy evaluation will be discussed.

7.5.1 Measuring the Accuracy of Ratings Prediction

Once the evaluation design for an offline experiment has been finalized, the accuracy needs to be measured over the test set. As discussed earlier, let S be the set of specified (observed) entries, and $E \subset S$ be the set of entries in the test set used for evaluation. Each entry in E is a user-item index pair of the form (u, j) corresponding to a position in the ratings matrix. Note that the set E may correspond to the held out entries in the hold-out method, or it may correspond to one of the partitions of size $|S|/q$ during cross-validation.

Let r_{uj} be the value of the (hidden) rating of entry $(u, j) \in E$, which is used in the test set. Furthermore, let $\hat{r}_{uj}$ be the predicted rating of the entry (u, j) by the specific training algorithm being used. The entry-specific error is given by $e_{uj} = \hat{r}_{uj} - r_{uj}$. This error can be leveraged in various ways to compute the overall error over the set E of entries on which the evaluation is performed. An example is the *mean squared error*, denoted by *MSE*:

$$MSE = \frac{\sum_{(u,j) \in E} e_{uj}^2}{|E|} \tag{7.4}$$

Clearly, smaller values of the *MSE* are indicative of superior performance. The square-root of this value is referred to as the *root mean squared error (RMSE)*, and it is often used instead of the *MSE*:

$$RMSE = \sqrt{\frac{\sum_{(u,j)\in E} e_{uj}^2}{|E|}} \tag{7.5}$$

The *RMSE* is in units of ratings, rather than in units of squared ratings like the *MSE*. The *RMSE* was used as the standard metric for the Netflix Prize contest. One characteristic of the *RMSE* is that it tends to disproportionately penalize large errors because of the squared term within the summation. One measure, known as the *mean-absolute-error (MAE)*, does not disproportionately penalize larger errors:

$$MAE = \frac{\sum_{(u,j)\in E} |e_{uj}|}{|E|} \tag{7.6}$$

Other related measures such as the normalized *RMSE* (*NRMSE*) and normalized *MAE* (*NMAE*) are defined in a similar way, except that each of them is divided by the range $r_{max} - r_{min}$ of the ratings:

$$NRMSE = \frac{RMSE}{r_{max} - r_{min}}$$
$$NMAE = \frac{MAE}{r_{max} - r_{min}}$$

The normalized values of the *RMSE* and *MAE* always lie in the range $(0, 1)$, and therefore they are more interpretable from an intuitive point of view. It is also possible to use these values to compare the performance of a particular algorithm over different data sets with varying scales of ratings.

7.5.1.1 RMSE versus MAE

Is *RMSE* or *MAE* better as an evaluation measure? There is no clear answer to this question, as this depends on the application at hand. As the *RMSE* sums up the squared errors, it is more significantly affected by large error values or outliers. A few badly predicted ratings can significantly ruin the *RMSE* measure. In applications where robustness of prediction across various ratings is very important, the *RMSE* may be a more appropriate measure. On the other hand, the *MAE* is a better reflection of the accuracy when the importance of outliers in the evaluation is limited. The main problem with *RMSE* is that it is not a true reflection of the average error, and it can sometimes lead to misleading results [632]. Clearly, the specific choice should depend on the application at hand. A discussion of the relative benefits of the two kinds of measures can be found in [141].

7.5.1.2 Impact of the Long Tail

One problem with these metrics is that they are heavily influenced by the ratings on the popular items. The items that receive very few ratings are ignored. As discussed in Chapter 2, ratings matrices exhibit a long-tail property, in which the vast majority of items are bought (or rated) rarely. We have replicated Figure 2.1 of Chapter 2 in Figure 7.3. The X-axis represents the indices of the items in decreasing order of popularity, and the Y-axis indicates the rating frequency. It is evident that only a few items receive a large number of ratings, whereas most of the remaining items receive few ratings. The latter constitute the long tail. Unfortunately, items in the long tail often contribute to the vast majority of profit for merchants [49]. As a result, the most important items often get weighted the least in the evaluation process. Furthermore, it is often much harder to predict the values of

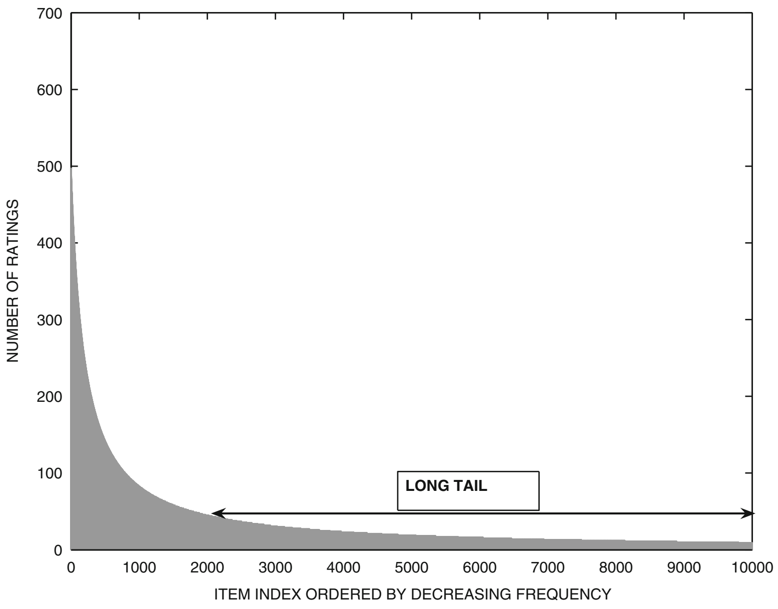


Figure 7.3: The long tail of rating frequencies (Revisiting Figure 2.1 of Chapter 2)

the ratings in the long tail because of greater local sparsity [173]. Therefore, the prediction accuracies on sparse items will typically be different from those on popular items. One way of handling this problem is to compute the *RMSE* or *MAE* separately for all the hidden ratings associated with each item, and then average over the different items in a weighted way. In other words, the accuracy computations of Equations 7.5 and 7.6 can be weighted with an item-specific weight, depending on the relative importance, profit, or utility to the merchant. It is also possible to perform these computations with user-specific weighting (rather than item-specific weighting), although the practical applicability of user-specific weighting is limited.

7.5.2 Evaluating Ranking via Correlation

The aforementioned measures are designed to evaluate the prediction accuracy of the actual rating value of a user-item combination. In practice, the recommender system creates a ranking of items for a user, and the top-k items are recommended. The value of k may vary with the system, item, and user at hand. In general, it is desirable for highly rated items to be ranked above items which are not highly rated. Consider a user u, for which the ratings of the set I_u of items have been hidden by a hold-out or cross-validation strategy. For example, if the ratings of the first, third, and fifth items (columns) of user (row) u are hidden for evaluation purposes, then we have $I_u = \{1, 3, 5\}$.

We would like to measure how well the ground-truth orderings of the ratings in I_u are related to the ordering predicted by the recommender system for the set I_u. An important issue to keep in mind is that ratings are typically chosen from a discrete scale, and many ties exist in the ground truth. Therefore, it is important for the ranking measures to not penalize the system for ranking one item above another when they are tied in the ground truth. The most common class of methods is to use *rank correlation coefficients*. The two most commonly used rank correlation coefficients are as follows:

1. *Spearman rank correlation coefficient:* The first step is to rank all items from 1 to $|I_u|$, both for the recommender system prediction and for the ground-truth. The Spearman correlation coefficient is simply equal to the Pearson correlation coefficient applied on these ranks. The computed value always ranges in $(-1, +1)$, and large positive values are more desirable.

   The Spearman correlation coefficient is specific to user u, and it can then be averaged over all users to obtain a global value. Alternatively, the Spearman rank correlation can be computed over all the hidden ratings over all users in one shot, rather than computing user-specific values and averaging them.

   One problem with this approach is that the ground truth will contain many ties, and therefore random tie-breaking might lead to some noise in the evaluation. For this purpose, an approach referred to as *tie-corrected Spearman* is used. One way of performing the correction is to use the average rank of all the ties, rather than using random tie-breaking. For example, if the ground-truth rating of the top-2 ratings is identical in a list of four items, then instead of using the ranks $\{1, 2, 3, 4\}$, one might use the ranks $\{1.5, 1.5, 3, 4\}$.

2. *Kendall rank correlation coefficient:* For each pair of items $j, k \in I_i$, the following credit $C(j, k)$ is computed by comparing the predicted ranking with the ground-truth ranking of these items:

$$C(j,k) = \begin{cases} +1 & \text{if items } j \text{ and } k \text{ are in the same relative order in} \\ & \text{ground-truth ranking and predicted ranking (concordant)} \\ -1 & \text{if items } j \text{ and } k \text{ are in a different relative order in} \\ & \text{ground-truth ranking and predicted ranking (discordant)} \\ 0 & \text{if items } j \text{ and } k \text{ are tied in either the} \\ & \text{ground-truth ranking or predicted ranking} \end{cases} \tag{7.7}$$

   Then, the Kendall rank correlation coefficient τ_u, which is specific to user u, is computed as the average value of $C(j, k)$ over all the $|I_u|(|I_u| - 1)/2$ pairs of test items for user i:

$$\tau_u = \frac{\sum_{j<k} C(j,k)}{|I_u| \cdot (|I_u| - 1)/2} \tag{7.8}$$

   A different way of understanding the Kendall rank correlation coefficient is as follows:

$$\tau_u = \frac{\text{Number of concordant pairs} - \text{Number of discordant pairs}}{\text{Number of pairs in } I_u} \tag{7.9}$$

   Note that this value is a *customer-specific* value of the Kendall coefficient. The value of τ_u may be averaged over all users u to obtain a heuristic global measure. Alternatively, one can perform the Kendall coefficient computation of Equation 7.8 over all hidden user-item pairs, rather than only the ones for customer u, in order to obtain a global value τ.

A number of other measures, such as the *normalized distance-based performance measure (NDPM)*, have been proposed in the literature. Refer to the bibliographic notes.

7.5.3 Evaluating Ranking via Utility

In the previous discussion, the ground-truth ranking is compared to the recommender system's ranking. Utility-based methods use the ground-truth *rating* in combination with the recommender system's ranking. For the case of implicit feedback data sets, the rating is substituted with a 0-1 value, depending on whether or not the customer has consumed the item. The overall goal of utility-based methods is to create a crisp quantification of how *useful* the customer might find the recommender system's ranking. An important principle underlying such methods is that recommendation lists are short compared to the total number of items. Therefore, most of the utility of a particular ranking should be based on the relevance of items, which are high in the recommended list. In this sense, the *RMSE* measure has a weakness because it equally weights the errors on the low-ranked items as compared to those on the highly-ranked items. It has been suggested [713] that small changes in *RMSE* such as 1%, can lead to large changes of more than 15% in the *identities* of the top-rated items. These are the only items that the end-user of the recommender system will actually see. Correspondingly, utility-based measures quantify the utility of a recommendation list by giving greater importance to the top-ranked items.

As in the previous sections, it is assumed that the ground-truth rating of each item in I_u is hidden from the recommender system before evaluation. Here, I_u represents the set of items rated by user u, which are hidden from the recommender system before evaluation. We will develop both user-specific and global utility quantifications.

In utility-based ranking, the basic idea is that each item in I_u has a utility to the user, which depends both on its position in the recommended list and its ground-truth rating. An item that has a higher ground-truth rating obviously has greater utility to the user. Furthermore, items ranked higher in the recommended list have greater utility to the user i because they are more likely to be noticed (by virtue of their position) and eventually selected. Ideally, one would like items with higher ground-truth rating to be placed as high on the recommendation list as possible.

How are these rating-based and ranking-based components defined? For any item $j \in I_u$, its rating-based utility to the user i is assumed to be $\max\{r_{uj} - C_u, 0\}$, where C_u is a break-even (neutral) rating value for user u. For example, C_u might be set to the mean rating of user u. On the other hand, the ranking-based utility of the item is $2^{-(v_j-1)/\alpha}$, where v_j is the rank of item j in the list of recommended items and α is a half-life parameter. In other words, the ranking-based utility exponentially decays with its rank, and moving down the ranks by α reduces the utility by a factor of 2. The logic of the decay-based ranking component is to ensure that the final utility of a particular ranking is regulated primarily by the top few items. After all, the user rarely browses the items that are very low in the list. The utility $F(u,j)$ of item $j \in I_u$ to user u is defined as the product of the rating-based and ranking-based utility values:

$$F(u,j) = \frac{\max\{r_{uj} - C_u, 0\}}{2^{(v_j-1)/\alpha}} \tag{7.10}$$

The R-score, which is specific to user u, is the sum of $F(u,j)$ over all the hidden ratings in I_u:

$$\text{R-score}(u) = \sum_{j \in I_u} F(u,j) \tag{7.11}$$

Note that the value of v_j can take on any value from 1 to n, where n is the total number of items. However, in practice, one often restricts the size of the recommended list to a maximum value of L. One can therefore compute the R-score over a recommended list of specific size L instead of using all the items, as follows:

$$\text{R-score}(u) = \sum_{j \in I_u, v_j \leq L} F(u, j) \tag{7.12}$$

The idea here is that ranks below L have no utility to the user because the recommended list is of size L. This variation is based on the principle that recommended lists are often very short compared to the total number of items. The overall R-score may be computed by summing this value over all the users.

$$\text{R-score} = \sum_{u=1}^{m} \text{R-score}(u) \tag{7.13}$$

The exponential decay in the utility implies that users are only interested in top-ranked items, and they do not pay much attention to lower-ranked items. This may not be true in all applications, especially in news recommender systems, where users typically browse multiple items lower down the list of recommended items. In such cases, the discount rate should be set in a milder way. An example of such a measure is the discounted cumulative gain (DCG). In this case, the discount factor of item j is set to $\log_2(v_j + 1)$, where v_j is the rank of item j in the test set I_u. Then, the discounted cumulative gain is defined as follows:

$$DCG = \frac{1}{m} \sum_{u=1}^{m} \sum_{j \in I_u} \frac{g_{uj}}{\log_2(v_j + 1)} \tag{7.14}$$

Here, g_{uj} represents the utility (or gain) of the user u in consuming item j. Typically, the value of g_{uj} is set to an exponential function of the relevance (e.g., non-negative ratings or user hit rates):

$$g_{uj} = 2^{rel_{uj}} - 1 \tag{7.15}$$

Here, rel_{uj} is the ground-truth relevance of item j for user u, which is computed as a heuristic function of the ratings or hits. In many settings, the raw ratings are used. It is common to compute the discounted cumulative gain over a recommendation list of specific size L, rather than using all the items:

$$DCG = \frac{1}{m} \sum_{u=1}^{m} \sum_{j \in I_u, v_j \leq L} \frac{g_{uj}}{\log_2(v_j + 1)} \tag{7.16}$$

The basic idea is that recommended lists have size no larger than L.

Then, the normalized discounted cumulative gain (NDCG) is defined as ratio of the discounted cumulative gain to its ideal value, which is also referred to as ideal discounted cumulative gain (IDCG).

$$NDCG = \frac{DCG}{IDCG} \tag{7.17}$$

The ideal discounted cumulative gain is computed by repeating the computation for DCG, except that the ground-truth rankings are used in the computation.

Another measure that is commonly used, is the average reciprocal hit rate (ARHR) [181]. This measure is designed for implicit feedback data sets, in which each value of $r_{uj} \in \{0, 1\}$. Therefore, a value of $r_{uj} = 1$ represents a "hit" where a customer has bought or clicked on an item. A value of $r_{uj} = 0$ corresponds to a situation where a customer has not bought or clicked on an item. In this implicit feedback setting, missing values in the ratings matrix are assumed to be 0.

In this case, the rank-based discount rate is $1/v_j$, where v_j is the rank of item j in the recommended list, and the item utility is simply the hidden "rating" value $r_{uj} \in \{0, 1\}$. Note that the discount rate is not as rapid as the R-score metric, but it is faster than DCG. Therefore, the combined utility of an item is given by r_{uj}/v_j. This expression represents the contribution of item $j \in I_u$ to the utility. Then, the ARHR metric for the user i is defined by summing up these values over all the hidden items in I_u:

$$ARHR(u) = \sum_{j \in I_u} \frac{r_{uj}}{v_j} \tag{7.18}$$

It is also possible to define the average reciprocal hit-rate for a recommended list of size L by adding only those utility values for which $v_j \leq L$.

$$ARHR(u) = \sum_{j \in I_u, v_j \leq L} \frac{r_{uj}}{v_j} \tag{7.19}$$

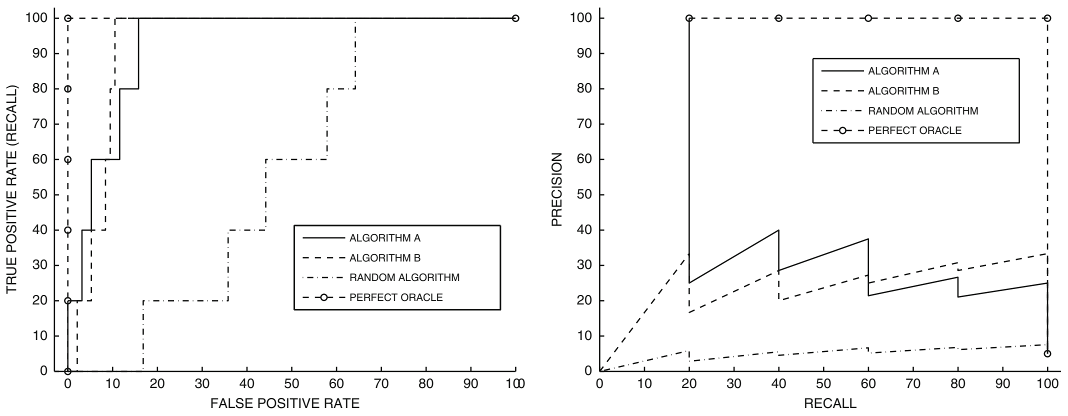
One quirk of the average reciprocal hit-rate is that it is typically used when the value of $|I_u|$ is exactly 1, and when the value r_{uj} of the corresponding (hidden) item $j \in I_u$ is always 1. Therefore, there is exactly one hidden item for each user, and the user has always bought or clicked on this item. In other words, the average reciprocal hit-rate rewards the utility (in a rank-reciprocal way) for recommending the single correct answer at a high position on the recommended list. This was the setting in which this measure was introduced [181], although one can generalize it to arbitrary settings in terms of the number of hidden items and explicit-feedback settings. The aforementioned expression provides this generalized definition because one can use a set I_u of arbitrary size in an explicit feedback setting. The global ARHR value is computed by averaging this value over the m users:

$$ARHR = \frac{\sum_{u=1}^{m} ARHR(u)}{m} \tag{7.20}$$

The average reciprocal hit-rate is also referred to as the *mean reciprocal rank (MRR).* In cases where the value of $|I_u|$ is 1, the average reciprocal hit-rate always lies in the range $(0, 1)$. In such cases, the hidden entry is usually an item for which $r_{uj} = 1$ and the length of the recommendation list is restricted to L. Note that only "hits" contribute to the utility in these cases. A simplification of this measure is the *hit-rate*, in which the rank-reciprocal weighting is not used, and the value of $|I_u|$ is exactly 1. Therefore, the hit-rate (HR) is simply the fraction of users for which the correct answer is included in the recommendation list of length L. The disadvantage of the hit-rate is that it gives equal importance to a hit, irrespective of its position in the recommended list.

The ARHR and HR are almost always used in implicit feedback data sets, in which missing values are treated as 0. Nevertheless, the definition of Equation 7.19 is stated in a more general way. Such a definition can also be used in the context of explicit feedback data sets, in which the values of r_{uj} need not be drawn from $\{0, 1\}$. In such cases, the ratings of any number of items of each user are hidden, and the values of the hidden ratings can be arbitrary. Furthermore, the missing values need not be treated as 0s, and I_u is always selected from the observed items.

A related measure is the *mean average precision (MAP),* which computes the fraction of relevant items in a recommended list of length L for a given user. Various equally spaced



(a) ROC (b)Precision-recall

Figure 7.4: ROC curve and precision-recall curves

values of L are used, and the precision is averaged over these recommendation lists of varying lengths. The resulting precision is then averaged over all the users.

Numerous other measures have been proposed in the literature to evaluate the effectiveness of rankings. For example, the *lift index* [361] divides the ranked items into deciles to compute a utility score. Refer to the bibliographic notes.

7.5.4 Evaluating Ranking via Receiver Operating Characteristic

Ranking methods are used frequently in the evaluation of the actual *consumption* of items. For example, Netflix might recommend a set of ranked items for a user, and the user might eventually consume only a subset of these items. Therefore, these methods are well suited to implicit feedback data sets, such as sales, click-throughs, or movie views. Such actions can be represented in the form of unary ratings matrices, in which missing values are considered to be equivalent to 0. Therefore, the ground-truth is of a binary nature.

The items that are eventually consumed are also referred to as the *ground-truth positives* or *true positives*. The recommendation algorithm can provide a ranked list of *any* number of items. What percentage of these items is relevant? A key issue here is that the answer to this question depends on the size of the recommended list. Changing the number of recommended items in the ranked list has a direct effect on the trade-off between the fraction of recommended items that are actually consumed and the fraction of consumed items that are captured by the recommender system. This trade-off can be measured in two different ways with the use of a precision-recall or a *receiver operating characteristic (ROC)* curve. Such trade-off plots are commonly used in rare class detection, outlier analysis evaluation, and information retrieval. In fact, such trade-off plots can be used in any application where a binary ground truth is compared to a ranked list discovered by an algorithm.

The basic assumption is that it is possible to rank all the items using a numerical score, which is the output of the algorithm at hand. Only the top items are recommended. By varying the size of the recommended list, one can then examine the fraction of relevant (ground-truth positive) items in the list, and the fraction of relevant items that are missed by the list. If the recommended list is too small, then the algorithm will miss relevant items (false-negatives). On the other hand, if a very large list is recommended, this will lead to too many spurious recommendations that are never used by the user (false-positives).

Table 7.1: Rank of ground-truth positive instances

Algorithm	**Rank of items that are truly used (ground-truth positives)**
Algorithm A	1, 5, 8, 15, 20
Algorithm B	3, 7, 11, 13, 15
Random Algorithm	17, 36, 45, 59, 66
Perfect Oracle	1, 2, 3, 4, 5

This leads to a trade-off between the false-positives and false-negatives. The problem is that the correct size of the recommendation list is never known exactly in a real scenario. However, the entire trade-off curve can be quantified using a variety of measures, and two algorithms can be compared over the entire trade-off curve. Two examples of such curves are the *precision-recall* curve and the *receiver operating characteristic (ROC)* curve.

Assume that one selects the top-t set of ranked items to recommend to the user. For any given value t of the size of the recommended list, the set of recommended items is denoted by $\mathcal{S}(t)$. Note that $|\mathcal{S}(t)| = t$. Therefore, as t changes, the size of $\mathcal{S}(t)$ changes as well. Let $\mathcal{G}$ represent the true set of relevant items (ground-truth positives) that are consumed by the user. Then, for any given size t of the recommended list, the *precision* is defined as the percentage of recommended items that truly turn out to be relevant (i.e., consumed by the user).

$$Precision(t) = 100 \cdot \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{S}(t)|}$$

The value of $Precision(t)$ is *not* necessarily monotonic in t because both the numerator and denominator may change with t differently. The *recall* is correspondingly defined as the percentage of *ground-truth* positives that have been recommended as positive for a list of size t.

$$Recall(t) = 100 \cdot \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{G}|}$$

While a natural trade-off exists between precision and recall, this trade-off is not necessarily monotonic. In other words, an increase in recall does not always lead to a reduction in precision. One way of creating a single measure that summarizes both precision and recall is the F_1-measure, which is the harmonic mean between the precision and the recall.

$$F_1(t) = \frac{2 \cdot Precision(t) \cdot Recall(t)}{Precision(t) + Recall(t)} \tag{7.21}$$

While the $F_1(t)$ measure provides a better quantification than either precision or recall, it is still dependent on the size t of the recommended list and is therefore still not a complete representation of the trade-off between precision and recall. It is possible to visually examine the entire trade-off between precision and recall by varying the value of t and plotting the precision versus the recall. As shown later with an example, the lack of monotonicity of the precision makes the results harder to intuitively interpret.

A second way of generating the trade-off in a more intuitive way is through the use of the ROC curve. The *true-positive rate*, which is the same as the recall, is defined as the percentage of ground-truth positives that have been included in the recommendation list of size t.

$$TPR(t) = Recall(t) = 100 \cdot \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{G}|}$$

The false-positive rate $FPR(t)$ is the percentage of the falsely reported positives in the recommended list out of the ground-truth negatives (i.e., irrelevant items not consumed by the user). Therefore, if $\mathcal{U}$ represents the universe of all items, the ground-truth negative set is given by $(\mathcal{U} - \mathcal{G})$, and the falsely reported part in the recommendation list is $(\mathcal{S}(t) - \mathcal{G})$. Therefore, the false-positive rate is defined as follows:

$$FPR(t) = 100 \cdot \frac{|\mathcal{S}(t) - \mathcal{G}|}{|\mathcal{U} - \mathcal{G}|} \tag{7.22}$$

The false-positive rate can be viewed as a kind of “bad” recall, in which the fraction of the ground-truth negatives (i.e., items not consumed), which are incorrectly captured in the recommended list $\mathcal{S}(t)$, is reported. The ROC curve is defined by plotting the $FPR(t)$ on the X-axis and $TPR(t)$ on the Y-axis for varying values of t. In other words, the ROC curve plots the “good” recall against the “bad” recall. Note that both forms of recall will be at 100% when $\mathcal{S}(t)$ is set to the entire universe of items. Therefore, the end points of the ROC curve are always at $(0, 0)$ and $(100, 100)$, and a random method is expected to exhibit performance along the diagonal line connecting these points. The *lift* obtained above this diagonal line provides an idea of the accuracy of the approach. The area under the ROC curve provides a concrete quantitative evaluation of the effectiveness of a particular method. Although one can directly use the area shown in Figure 7.4(a), the staircase-like ROC curve is often modified to use local linear segments which are not parallel to either the X-axis or the Y-axis. The trapezoidal rule [195] is then used to compute the area slightly more accurately. From a practical point of view, this change often makes very little difference to the final computation.

To illustrate the insights gained from these different graphical representations, consider an example of a scenario with 100 items, in which 5 items are truly relevant. Two algorithms A and B are applied to this data set that rank all items from 1 to 100, with lower ranks being selected first in the recommended list. Thus, the true-positive rate and false-positive rate values can be generated from the ranks of the 5 relevant items. In Table 7.1, some hypothetical ranks for the 5 truly relevant items have been illustrated for the different algorithms. In addition, the ranks of the ground-truth positive items for a random algorithm have been indicated. This algorithm ranks all the items randomly. Similarly, the ranks for a “perfect oracle” algorithm, which ranks the correct top 5 items in the recommended list, have also been illustrated in the table. The resulting ROC curves are illustrated in Figure 7.4(a). The corresponding precision-recall curves are illustrated in Figure 7.4(b). Note that the ROC curves are always increasing monotonically, whereas the precision-recall curves are not monotonic. While the precision-recall curves are not quite as nicely interpretable as the ROC curves, it is easy to see that the *relative trends* between different algorithms are the same in both cases. In general, ROC curves are used more frequently because of greater ease in interpretability.

What do these curves really tell us? For cases in which one curve strictly dominates another, it is clear that the algorithm for the former curve is superior. For example, it is immediately evident that the oracle algorithm is superior to all algorithms and that the random algorithm is inferior to all the other algorithms. On the other hand, algorithms A and B show domination at different parts of the ROC curve. In such cases, it is hard to say that one algorithm is strictly superior. From Table 7.1, it is clear that Algorithm A ranks three relevant items very highly, but the remaining two items are ranked poorly. In the case of Algorithm B, the highest ranked items are not as well ranked as Algorithm A, though all 5 relevant items are determined much earlier in terms of rank threshold. Correspondingly, Algorithm A dominates on the earlier part of the ROC curve, whereas

Algorithm B dominates on the later part. It is possible to use the area under the ROC curve as a proxy for the overall effectiveness of the algorithm. However, not all parts of the ROC curve are equally important because there are usually practical limits on the size of the recommended list.

The aforementioned description illustrates the generation of *customer-specific ROC curves*, because the ROC curves are specific to each user. It is also possible to generate *global ROC curves* by ranking user-item pairs and then using the same approach as discussed above. In order to rank user-item pairs, it is assumed that the algorithm has a mechanism to rank them by using predicted affinity values. For example, the predicted ratings for user-item pairs can be used to rank them.

7.5.5 Which Ranking Measure is Best?

Although ROC curves are often used for evaluating recommender systems, they do not always reflect the performance from the end-user perspective. In many settings, the end user sees only a small subset of top-ranked items. Measures such as ROC and Kendall coefficient, which treat higher and lower ranked items equally, are unable to capture the greater importance of higher ranked items. For example, the relative ranking between two items ranked first and second on the recommendation list is far more important than the relative ranking of two items, which are ranked 100th and 101st on the list. In this context, utility-based measures such as NDCG do a much better job than rank-correlation coefficients or ROC measures at distinguishing between higher-ranked and lower-ranked items.

7.6 Limitations of Evaluation Measures

Accuracy-based evaluation measures have a number of weaknesses that arise out of selection bias in recommender systems. In particular, the missing entries in a ratings matrix are not random because users have the tendency of rating more popular items. As shown in Figure 7.3, a few items are rated by many users, whereas the vast majority of items may be found in the *long tail*. The distributions of the ratings on popular items are often different from those on items in the long tail. When an item is very popular, it is most likely because of the notable content in it. This factor will affect[2] the rating of that item as well. As a result, the accuracy of most recommendation algorithms is different on the more popular items versus the items in the long tail [564]. More generally, the fact that a particular user has *chosen* not to rate a particular item thus far has a significant impact on what her rating would be if the user were forced to rate all items. This issue is stated in [184] in a somewhat different context as follows:

> "Intuitively, a simple process could explain the results: users chose to rate songs they listen to, and listen to music they expect to like, while avoiding genres they dislike. Therefore, most of the songs that would get a bad rating are not voluntarily rated by the users. Since people rarely listen to random songs, or rarely watch random movies, we should expect to observe in many areas a difference between the distribution of ratings for random items and the corresponding distribution for the items selected by the users."

[2]A related effect is that observed ratings are likely to be specified by users who are frequent raters. Frequent raters may show different patterns of rating values compared to infrequent raters.

These factors cause problems of bias in the evaluation process. After all, in order to perform the evaluation on a given data set, one cannot use truly missing ratings; rather, one must simulate missing items with the use of hold-out or cross-validation mechanisms on ratings that are already specified. Therefore, the *simulated* missing items may not show similar accuracy to that one would theoretically obtain on the *truly* consumed items in the future. The items that are consumed in the future will not be randomly selected from the missing entries for the reasons discussed above. This property of rating distributions is also known as *Missing Not At Random (MNAR)*, or *selection bias* [402, 565]. This property can lead to an incorrect *relative* evaluation of algorithms. For example, a popularity-based model in which items with the highest mean rating are recommended might do better in terms of gaining more revenue for the merchant than its evaluation on the basis of randomly missing ratings might suggest. This problem is aggravated by the fact that items in the long tail are especially important to the recommender system, because a disproportionate portion of the profits in such systems are realized through such items.

There are several solutions to this issue. The simplest solution is to not select the missing ratings at random but to use a model for selecting the test ratings based on their likelihood of being rated in the future. Another solution is to not divide the ratings at random between training and test, but to divide them *temporally* by using more recent ratings as a part of the test data; indeed, the Netflix Prize contest used more recent ratings in the qualifying set, although some of the recent ratings were also provided as a part of the probe set. An approach that has been used in recent years, is to correct for this bias by modeling the bias in the missing rating distribution within the evaluation measures [565, 566]. Although such an approach has some merits, it does have the drawback that the evaluation process itself now *assumes* a model of how the ratings behave. Such an approach might inadvertently favor algorithms that use a model similar to that used for the prediction of ratings as for the evaluation process. It is noteworthy that many recent algorithms [309] use implicit feedback within the *prediction* process. This raises the possibility that a future *prediction* algorithm might be designed to be tailored to the model used for adjusting for the effect of user selection bias within the *evaluation*. Although the assumptions in [565], which relate the missing ratings to their relevance, are quite reasonable, the addition of more assumptions (or complexity) to evaluation mechanisms increases the possibility of "gaming" during benchmarking. At the end of the day, it is important to realize that these limitations in collaborative filtering evaluation are inherent; the quality of any evaluation system is fundamentally limited by the quality of the available ground truth. In general, it has been shown through experiments on Netflix data [309] that the use of straightforward RMSE measures on the observed ratings often correlate quite well with the precision on all items.

Another source of evaluation bias is the fact that user interests may evolve with time. As a result, the performance on a hold-out set might not reflect future performance. Although it is not a perfect solution, the use of *temporal* divisions between training and test ratings seems like a reasonable choice. Even though temporal division results in training and testing tests with somewhat different distributions, it also reflects the real-world setting more closely. In this sense, the Netflix Prize contest again provides an excellent model of realistic evaluation design. Several other variations of temporal methods in the evaluation process are discussed in [335].

7.6.1 Avoiding Evaluation Gaming

The fact that missing ratings are not random can sometimes lead to unintended (or intended) gaming of the evaluations in settings where the user-item pairs of the test entries are specified. For example, in the Netflix Prize contest, the *coordinates* of the user-item pairs in the qualifying set were specified, although the *values* of the ratings were not specified. By incorporating the coordinates of the user-item pairs within the qualifying set as implicit feedback (i.e., matrix F in section 3.6.4.6), one can improve the quality of recommendations. It can be argued that such an algorithm would have an unfair advantage over one that did not include any information about the identities of rated items in the qualifying set. The reason is that in real-life settings, one would never have any information about future coordinates of rated items, as are easily available in the qualifying set of the Netflix Prize data. Therefore, the additional advantage of incorporating such implicit feedback would disappear in real-life settings. One solution would be to not specify the coordinates of test entries and thereby evaluate over all entries. However, if the ratings matrix has very large dimensions (e.g., $10^7 \times 10^5$), it may be impractical to perform the prediction over all entries. Furthermore, it would be difficult to store and upload such a large number of predictions in an online contest like the Netflix Prize. In such cases, an alternative would be to include (spurious) unrated entries within the test set. Such entries are not used for evaluation but they have the effect of preventing the use of coordinates of the test entries as implicit feedback.

7.7 Summary

The evaluation of recommender systems is crucial in order to obtain a clear idea about the quality of different algorithms. The most direct method of measuring the effectiveness of a recommender system is to compute the conversion rate at which recommended items are converted to actual usages. This can be done through either user studies or online studies. Such studies are often difficult for researchers and practitioners because of the difficulty in obtaining access to the relevant infrastructure with large groups of users. Offline methods have the advantage that they can be used with multiple historical data sets. In such cases, it is dangerous to use accuracy as the only criterion, because maximizing accuracy does not always lead to long-term maximization of conversion rates. A variety of criteria, such as coverage, novelty, serendipity, stability, and scalability, are used to evaluate the effectiveness of recommender systems.

The proper design of recommender evaluation systems is necessary to ensure that there are no biases in the evaluation process. For example, in a collaborative filtering application, it is important to ensure that all the ratings are evaluated with an out-of-sample approach. A variety of methods such as hold-out and cross-validation are used in order to ensure out-of-sample evaluation. The error is computed with measures, such as the MAE, MSE, and RMSE. In some measures, items are weighted differently to account for their differential importance. In order to evaluate the effectiveness of ranking methods, rank correlation, utility-based measures or usage-based measures may be used. For usage-based measures, precision and recall are used to characterize the trade-off inherent in varying the size of the recommended list. The F1-measure is also used, which is the harmonic mean between the precision and the recall.

7.8 Bibliographic Notes

Excellent discussions on evaluating recommender systems may be found in [246, 275, 538]. Evaluation can be performed either with user studies or with historical data sets. The earliest work on evaluation with user studies may be found in [339, 385, 433]. An early study of evaluation of recommendation algorithms with historical data sets may be found in [98]. Metrics for evaluating recommender systems in the presence of cold-start are discussed in [533]. Controlled experiments for online evaluation in Web applications are discussed in [305]. A general study of online evaluation design is provided in [93]. The evaluation of multi-armed bandit systems is discussed in [349]. A comparison of online recommender systems with respect to human decisions is provided in [317].

The work in [246] presents several variants of accuracy metrics for evaluation. This article is perhaps one of the foremost authorities on the evaluation of recommender systems. The pitfalls of using the *RMSE* as an evaluation measure are presented in [632]. A brief technical note on the relative merits of using *MAE* and *RMSE* as accuracy measures may be found in [141]. The challenges and pitfalls in the use of accuracy metrics are discussed in [418]. Alternative methods for evaluating recommender systems are provided in [459]. A discussion of the importance of novelty is provided in [308]. Online methods for measuring the novelty of a recommender system are provided in [140, 286]. The use of popularity in the measurement of novelty is discussed in [140, 539, 680]. The work in [670] showed that serendipity can be achieved in a recommender system with the help of user labeling. Metrics for serendipity evaluation are discussed in [214, 450]. The work in [214] also studies the use of coverage metrics. Diversity metrics are discussed in [560]. The impact of recommender systems on sales diversity is discussed in [203]. Robustness and stability metrics for recommender systems are discussed in [158, 329, 393, 444]. A study of the evaluation of classification systems may be found in [18, 22]. The discussions in these books provide an understanding of the standard techniques used, such as hold-out and cross-validation.

Rank correlation methods are discussed in [298, 299]. The normalized distance preference measure is discussed in [505]. The R-score for the utility-based evaluation of rankings is discussed in [98]. The NDCG measure is discussed in [59]. The lift index is discussed in [361], whereas the average reciprocal hit rate (ARHR) is proposed in [181]. A discussion of ROC curves in the context of classification may be found in [195], although the same ideas are also applicable to the case of recommender systems. The use of customer-specific and global ROC curves is discussed in [533].

One limitation of recommender systems is that the values on the ratings are related to their relative frequency and that missing items are often in the long tail. Therefore, the use of cross-validation or hold-out mechanisms leads to a selection bias against less frequent items. A number of recent methods for correcting for missing item bias are discussed in [402, 564–566]. The approach in [565] proposes the use of different assumptions for the relevant and non-relevant items, in terms of deciding which ratings are missing. A training algorithm is also designed in [565] based on these assumptions. A temporal framework for realistic evaluation is discussed in [335]. Recommender systems also need to be evaluated somewhat differently in various settings, such as in the presence of specific contexts. These contexts could include time, location, or social information. An evaluation framework for recommender systems in the context of temporal data is provided in [130]. A recent workshop that was devoted exclusively to recommender systems' evaluation may be found in [4].

7.9 Exercises

1. Suppose that a merchant knows the amount of profit q_i made on the sale of the ith item. Design an error metric for a collaborative filtering system that weights the importance of each item with its profit.

2. Suppose that you designed an algorithm for collaborative filtering and found that it was performing poorly on ratings with value 5, but it was performing well on the other ratings. You used this insight to modify your algorithm and tested the algorithm again. Discuss the pitfalls with the second evaluation. Relate your answer to why Netflix chose to separate the quiz set and the test set in the Netflix Prize data set.

3. Implement an algorithm for constructing the ROC and the precision-recall curves.

4. Suppose you have an implicit feedback data set in which the ratings are unary. Would an ROC curve provide more meaningful results or would the *RMSE* metric?

5. Consider a user John, for whom you have hidden his ratings for *Aliens* (5), *Terminator* (5), *Nero* (1), and *Gladiator* (6). The values in brackets represent his hidden ratings, and higher values are better. Now consider a scenario where the recommender system ranks these movies in the order *Terminator*, *Aliens*, *Gladiator*, *Nero*.

(a) Compute the Spearman rank correlation coefficient as a measure of recommendation ranking quality.

(b) Compute the Kendall rank correlation coefficient as a measure of ranking quality.

6. For the problem in Exercise 5, John's utility for a movie j is given by $\max\{r_{ij} - 3, 0\}$, where r_{ij} is his rating.

(a) Under this utility assumption, compute the R-score specific to John. Assume a half-life value of $\alpha = 1$.

(b) For the same utility assumption, compute the component of the discounted cumulative gain (DCG) specific to John, if there are a total of 10 users in the system.

7. For the problem in Exercise 5, assume that the only hidden ratings belong to John, and the predicted ratings from the recommender system are *Aliens* (4.3), *Terminator* (5.4), *Nero* (1.3), and *Gladiator* (5). The values in brackets represent the predicted ratings.

(a) Compute the MSE of the predicted ratings.

(b) Compute the MAE of the predicted ratings.

(c) Compute the RMSE of the predicted ratings.

(d) Compute the normalized MAE and RMSE, assuming that all ratings lie in the range $\{1 \dots 6\}$.