4

Knowledge-based recommendation

4.1 Introduction

Most commercial recommender systems in practice are based on collaborative filtering (CF) techniques, as described in Chapter 2. CF systems rely solely on the user ratings (and sometimes on demographic information) as the only knowledge sources for generating item proposals for their users. Thus, no additional knowledge – such as information about the available movies and their characteristics – has to be entered and maintained in the system.

Content-based recommendation techniques, as described in Chapter 3, use different knowledge sources to make predictions whether a user will like an item. The major knowledge sources exploited by content-based systems include category and genre information, as well as keywords that can often be automatically extracted from textual item descriptions. Similar to CF, a major advantage of content-based recommendation methods is the comparably low cost for knowledge acquisition and maintenance.

Both collaborative and content-based recommender algorithms have their advantages and strengths. However, there are many situations for which these approaches are not the best choice. Typically, we do not buy a house, a car, or a computer very frequently. In such a scenario, a pure CF system will not perform well because of the low number of available ratings (Burke 2000). Furthermore, time spans play an important role. For example, five-year-old ratings for computers might be rather inappropriate for content-based recommendation. The same is true for items such as cars or houses, as user preferences evolve over time because of, for example, changes in lifestyles or family situations. Finally, in more complex product domains such as cars, customers often want to define their requirements explicitly – for example, "the maximum price of the car is x and the color should be black". The formulation of such requirements is not typical for pure collaborative and content-based recommendation frameworks.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Knowledge-based recommender systems help us tackle the aforementioned challenges. The advantage of these systems is that no ramp-up problems exist, because no rating data are needed for the calculation of recommendations. Recommendations are calculated independently of individual user ratings: either in the form of *similarities* between customer requirements and items or on the basis of explicit *recommendation rules*. Traditional interpretations of what a recommender system is focus on the *information filtering* aspect (Konstan et al. 1997, Pazzani 1999a), in which items that are likely to be of interest for a certain customer are filtered out. In contrast, the recommendation process of knowledge-based recommender applications is highly interactive, a foundational property that is a reason for their characterization as *conversational systems* (Burke 2000). This interactivity aspect triggered a slight shift from the interpretation as a filtering system toward a wider interpretation where recommenders are defined as systems that "guide a user in a personalized way to interesting or useful objects in a large space of possible options or that produce such objects as output" (Burke 2000). Recommenders that rely on knowledge sources not exploited by collaborative and content-based approaches are by default defined as knowledge-based recommenders by Burke (2000) and Felfernig and Burke (2008).

Two basic types of knowledge-based recommender systems are *constraint-based* (Felfernig and Burke 2008, Felfernig et al. 2006–07, Zanker et al. 2010) and *case-based* systems (Bridge et al. 2005, Burke 2000). Both approaches are similar in terms of the recommendation process: the user must specify the requirements, and the system tries to identify a solution. If no solution can be found, the user must change the requirements. The system may also provide explanations for the recommended items. These recommenders, however, differ in the way they use the provided knowledge: case-based recommenders focus on the retrieval of similar items on the basis of different types of similarity measures, whereas constraint-based recommenders rely on an explicitly defined set of recommendation rules. In constraint-based systems, the set of recommended items is determined by, for instance, searching for a set of items that fulfill the recommendation rules. Case-based systems, on the other hand, use similarity metrics to retrieve items that are similar (within a predefined threshold) to the specified customer requirements. Constraint-based and case-based knowledge representations will be discussed in the following subsections.

4.2 Knowledge representation and reasoning

In general, knowledge-based systems rely on detailed knowledge about item characteristics. A snapshot of such an item catalog is shown in Table 4.1 for

Table 4.1. *Example product assortment: digital cameras (Felfernig et al. 2009).*

id	price(€)	mpix	opt-zoom	LCD-size	movies	sound	waterproof
p_1	148	8.0	4×	2.5	no	no	yes
p_2	182	8.0	5×	2.7	yes	yes	no
p_3	189	8.0	10×	2.5	yes	yes	no
p_4	196	10.0	12×	2.7	yes	no	yes
p_5	151	7.1	3×	3.0	yes	yes	no
p_6	199	9.0	3×	3.0	yes	yes	no
p_7	259	10.0	3×	3.0	yes	yes	no
p_8	278	9.1	10×	3.0	yes	yes	yes

the digital camera domain. Roughly speaking, the recommendation problem consists of selecting items from this catalog that match the user's needs, preferences, or hard requirements. The user's requirements can, for instance, be expressed in terms of desired values or value ranges for an item feature, such as "the price should be lower than 300€" or in terms of desired functionality, such as "the camera should be suited for sports photography".

Following the categorization from the previous section, we now discuss how the required domain knowledge is encoded in typical knowledge-based recommender systems. A constraint-based recommendation problem can, in general, be represented as a *constraint satisfaction problem* (Felfernig and Burke 2008, Zanker et al. 2010) that can be solved by a constraint solver or in the form of a *conjunctive query* (Jannach 2006a) that is executed and solved by a database engine. Case-based recommendation systems mostly exploit similarity metrics for the retrieval of items from a catalog.

4.2.1 Constraints

A classical constraint satisfaction problem (CSP)[1] can be described by a-tuple (V, D, C) where

- V is a set of variables,
- D is a set of finite domains for these variables, and
- C is a set of constraints that describes the combinations of values the variables can simultaneously take (Tsang 1993).

A solution to a CSP corresponds to an assignment of a value to each variable in V in a way that all constraints are satisfied.

[1] A discussion of different CSP algorithms can be found in Tsang (1993).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.2. *Example recommendation task (V_C, V_{PROD}, C_R, C_F, C_{PROD}, REQ) and the corresponding recommendation result (RES).*

V_C	{*max-price*(0 . . . 1000), *usage*(*digital, small-print, large-print*), *photography* (*sports*, *landscape*, *portrait*, *macro*)}
V_{PROD}	{*price*(0 . . . 1000), *mpix*(3.0 . . . 12.0), *opt-zoom*(4× . . . 12×), *lcd-size* (2.5 . . . 3.0), *movies*(*yes*, *no*), *sound*(*yes*, *no*), *waterproof*(*yes*, *no*)}
C_F	{*usage* = *large-print* → *mpix* > 5.0} (*usage* is a customer property and *mpix* is a product property)
C_R	{*usage* = *large-print* → *max-price* > *200*} (*usage* and *max-price* are customer properties)
C_{PROD}	{(*id*=p1 ∧ *price*=148 ∧ *mpix*=8.0 ∧ *opt-zoom*=4× ∧ *lcd-size*=2.5 ∧ *movies*=*no* ∧ *sound*=*no* ∧ *waterproof*=no) ∨ · · · ∨ (*id*=p8 ∧ *price*=278 ∧ *mpix*=9.1 ∧ *opt-zoom*=10× ∧ *lcd-size*=3.0 ∧ *movies*=*yes* ∧ *sound*=*yes* ∧ *waterproof*=yes)}
REQ	{*max-price* = 300, *usage* = *large-print*, *photography* = *sports*}
RES	{*max-price* = 300, *usage* = *large-print*, *photography* = *sports*, *id* = p8, *price*=278, *mpix*=9.1, *opt-zoom*=10×, *lcd-size*=3.0, *movies*=*yes*, *sound*=*yes*, *waterproof*=yes}

Constraint-based recommender systems (Felfernig and Burke 2008, Felfernig et al. 2006–07, Zanker et al. 2010) can build on this formalism and exploit a *recommender knowledge base* that typically includes two different sets of variables ($V = V_C \cup V_{PROD}$), one describing potential customer requirements and the other describing product properties. Three different sets of constraints ($C = C_R \cup C_F \cup C_{PROD}$) define which items should be recommended to a customer in which situation. Examples for such variables and constraints for a digital camera recommender, as described by Jannach (2004), and Felfernig et al. (2006–07), are shown in Table 4.2.

- *Customer properties (V_C)* describe the possible customer requirements (see Table 4.2). The customer property *max-price* denotes the maximum price acceptable for the customer, the property *usage* denotes the planned usage of photos (print versus digital organization), and *photography* denotes the predominant type of photos to be taken; categories are, for example, sports or portrait photos.
- *Product properties (V_{PROD})* describe the properties of products in an assortment (see Table 4.2); for example, *mpix* denotes possible resolutions of a digital camera.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

- *Compatibility constraints (C_R)* define allowed instantiations of customer properties – for example, *if large-size photoprints are required, the maximal accepted price must be higher than 200* (see Table 4.2).
- *Filter conditions (C_F)* define under which conditions which products should be selected – in other words, filter conditions define the relationships between customer properties and product properties. An example filter condition is *large-size photoprints require resolutions greater than 5 mpix* (see Table 4.2).
- *Product constraints (C_{PROD})* define the currently available product assortment. An example constraint defining such a product assortment is depicted in Table 4.2. Each conjunction in this constraint completely defines a product (item) – all product properties have a defined value.

The task of identifying a set of products matching a customer's wishes and needs is denoted as a *recommendation task*. The customer requirements *REQ* can be encoded as unary constraints over the variables in V_C and V_{PROD} – for example, *max-price* $= 300$.

Formally, each solution to the CSP ($V = V_C \cup V_{PROD}$, D, $C = C_R \cup C_F \cup C_{PROD} \cup REQ$) corresponds to a consistent recommendation. In many practical settings, the variables in V_C do not have to be instantiated, as the relevant variables are already bound to values through the constraints in *REQ*. The task of finding such valid instantiations for a given constraint problem can be accomplished by every standard constraint solver. A consistent recommendation *RES* for our example recommendation task is depicted in Table 4.2.

Conjunctive queries. A slightly different way of constraint-based item retrieval for a given catalog, as shown in Table 4.1, is to view the item selection problem as a data filtering task. The main task in such an approach, therefore, is not to find valid variable instantiations for a CSP but rather to construct a conjunctive database query that is executed against the item catalog. A *conjunctive query* is a database query with a set of selection criteria that are connected conjunctively.

For example, $\sigma_{[mpix \geq 10, price < 300]}(P)$ is such a conjunctive query on the database table P, where σ represents the selection operator and $[mpix \geq 10, price < 300]$ the corresponding selection criteria. If we exploit conjunctive queries (database queries) for item selection purposes, V_{PROD} and C_{PROD} are represented by a database table P. Table attributes represent the elements of V_{PROD} and the table entries represent the constraint(s) in C_{PROD}. In our working example, the set of available items is $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ (see Table 4.1).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Queries can be defined that select different item subsets from P depending on the requirements in REQ. Such queries are directly derived from the filter conditions (C_F) that define the relationship between customer requirements and the corresponding item properties. For example, the filter condition $usage = large\text{-}print \rightarrow mpix > 5.0$ denotes the fact that if customers want to have large photoprints, the resolution of the corresponding camera ($mpix$) must be > 5.0. If a customer defines the requirement $usage = large\text{-}print$, the corresponding filter condition is active, and the consequent part of the condition will be integrated in a corresponding conjunctive query. The existence of a recommendation for a given set REQ and a product assortment P is checked by querying P with the derived conditions (consequents of filter conditions). Such queries are defined in terms of selections on P formulated as $\sigma_{[criteria]}(P)$, for example, $\sigma_{[mpix \geq 10]}(P) = \{p_4, p_7\}$.[2]

4.2.2 Cases and similarities

In case-based recommendation approaches, items are retrieved using similarity measures that describe to which extent item properties match some given user's requirements. The so-called distance similarity (McSherry 2003a) of an item p to the requirements $r \in REQ$ is often defined as shown in Formula 4.1. In this context, $sim(p, r)$ expresses for each *item attribute value* $\phi_r(p)$ its distance to the customer requirement $r \in REQ$ – for example, $\phi_{mpix}(p_1) = 8.0$. Furthermore, w_r is the importance weight for requirement r.[3]

$$similarity(p, REQ) = \frac{\sum_{r \in REQ} w_r * sim(p, r)}{\sum_{r \in REQ} w_r} \tag{4.1}$$

In real-world scenarios, there are properties a customer would like to maximize – for example, the resolution of a digital camera. There are also properties that customers want to minimize – for example, the price of a digital camera or the risk level of a financial service. In the first case we are talking about "more-is-better" (MIB) properties; in the second case the corresponding properties are denoted with "less-is-better" (LIB).

To take those basic properties into account in our similarity calculations, we introduce the following formulae for calculating local similarities

[2] For reasons of simplicity in the following sections we assume $V_C = V_{PROD}$ – that is, customer requirements are directly defined on the technical product properties. Queries on a product table P will be then written as $\sigma_{[REQ]}(P)$.

[3] A detailed overview of different types of similarity measures can be found in Wilson and Martinez 1997. Basic approaches to determine the importance of requirements (w) are discussed in Section 4.3.4.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

(McSherry 2003a). First, in the case of MIB properties, the local similarity between p and r is calculated as follows:

$$sim(p, r) = \frac{\phi_r(p) - min(r)}{max(r) - min(r)} \tag{4.2}$$

The local similarity between p and r in the case of LIB properties is calculated as follows:

$$sim(p, r) = \frac{max(r) - \phi_r(p)}{max(r) - min(r)} \tag{4.3}$$

Finally, there are situations in which the similarity should be based solely on the distance to the originally defined requirements. For example, if the user has a certain run time of a financial service in mind or requires a certain monitor size, the shortest run time as well as the largest monitor will not represent an optimal solution. For such cases we have to introduce a third type of local similarity function:

$$sim(p, r) = 1 - \frac{|\phi_r(p) - r|}{max(r) - min(r)} \tag{4.4}$$

The similarity measures discussed in this section are often the basis for different case-based recommendation systems, which will be discussed in detail in Section 4.4. *Utility-based recommendation* – as, for instance, mentioned by Burke (2000) – can be interpreted as a specific type of knowledge-based recommendation. However, this approach is typically applied in combination with constraint-based recommendation (Felfernig et al. 2006–07) and sometimes as well, in combination with case-based recommenders (Reilly et al. 2007b). Therefore, this approach will be discussed in Section 4.3.4 as a specific functionality in the context of constraint-based recommendation.

4.3 Interacting with constraint-based recommenders

The general interaction flow of a knowledge-based, conversational recommender can be summarized as follows.

- The user specifies his or her initial preferences – for example, by using a web-based form. Such forms can be identical for all users or personalized to the specific situation of the current user. Some systems use a question/answer preference elicitation process, in which the questions can be asked either all at once or incrementally in a wizard-style, interactive dialog, as described by Felfernig et al. (2006–07).

- When enough information about the user's requirements and preferences has been collected, the user is presented with a set of matching items. Optionally, the user can ask for an explanation as to why a certain item was recommended.
- The user might revise his or her requirements, for instance, to see alternative solutions or narrow down the number of matching items.

Although this general user interaction scheme appears to be rather simple in the first place, practical applications are typically required to implement more elaborate interaction patterns to support the end user in the recommendation process. Think, for instance, of situations in which none of the items in the catalog satisfies all user requirements. In such situations, a conversational recommender should intelligently support the end user in resolving the problem and, for example, proactively propose some action alternatives.

In this section we analyze in detail different techniques to support users in the interaction with constraint-based recommender applications. These techniques help improve the usability of these applications and achieve higher user acceptance in dimensions such as trust or satisfaction with the recommendation process and the output quality (Felfernig et al. 2006–07).

4.3.1 Defaults

Proposing default values. Defaults are an important means to support customers in the requirements specification process, especially in situations in which they are unsure about which option to select or simply do not know technical details (Huffman and Kahn 1998). Defaults can support customers in choosing a reasonable alternative (an alternative that realistically fits the current preferences). For example, if a customer is interested in printing large-format pictures from digital images, the camera should support a resolution of more than 5.0 megapixels (default). The negative side of the coin is that defaults can also be abused to manipulate consumers to choose certain options. For example, users can be stimulated to buy a park distance control functionality in a car by presenting the corresponding default value (Herrmann et al. 2007). Defaults can be specified in various ways:

- *Static defaults*: In this case, one default is specified per customer property – for example, *default*(*usage*)=*large-print*, because typically users want to generate posters from high-quality pictures.
- *Dependent defaults*: In this case a default is defined on different combinations of potential customer requirements – for example, *default*(*usage*=*small-print*, *max-price*) = 300.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.3. *Example of customer interaction data.*

customer (user)	price	opt-zoom	lcd-size
cu_1	400	10×	3.0
cu_2	300	10×	3.0
cu_3	150	4×	2.5
cu_4	200	5×	2.7
cu_5	200	5×	2.7

- *Derived defaults*: When the first two default types are strictly based on a declarative approach, this third type exploits existing interaction logs for the automated derivation of default values.

The following example sketches the main idea and a basic scheme for derived default values. Assume we are given the sample interaction log in Table 4.3. The only currently known requirement of a new user should be *price=400*; the task is to find a suitable default value for the customer requirement on the optical zoom (*opt-zoom*). From the interaction log we see that there exists a customer (cu_1) who had similar requirements (*price=400*). Thus, we could take cu_1's choice for the optical zoom as a default also for the new user.

Derived defaults can be determined based on various schemes; basic example approaches to the determination of suitable default values are, for example, *1-nearest neighbor* and *weighted majority voter*.

- *1-Nearest neighbor*: The 1-nearest neighbor approach can be used for the prediction of values for one or a set of properties in V_C. The basic idea is to determine the entry of the interaction log that is as close as possible to the set of requirements (*REQ*) specified by the customer. The 1-nearest neighbor is the entry in the example log in Table 4.3 that is most similar to the customer requirements in *REQ* (see Formula 4.1). In our working example, the nearest neighbor for the set of requirements $REQ = \{r_1 : price = 400, r_2 : opt\text{-}zoom = 10\times\}$ would be the interaction log entry for customer cu_1. If, for example, the variable *lcd-size* is not specified by the current customer, the recommender application could propose the value 3.0.
- *Weighted majority voter*: The weighted majority voter proposes customer property values that are based on the voting of a set of neighbor items for a specific property. It operates on a set of n-nearest neighbors, which can be calculated on the basis of Formula 4.1. Let us assume that the three-nearest neighbors for the requirements $REQ = \{r_1 : price = 400\}$ are the interaction

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

log entries for the customers $\{cu_1, cu_2, cu_4\}$ and we want to determine a default for the property *opt-zoom*. The majority value for *opt-zoom* would then be $10\times$, which, in this context, can be recommended as the default.

For weighted majority voters as well as for simple 1-nearest-neighbor-based default recommendations, it is not possible to guarantee that the requirements (including the defaults) allow the derivation of a recommendation. For example, if $REQ = \{r_1 : opt\text{-}zoom = 3\times\}$ then the weighted majority voter approach would recommend $lcd\text{-}size = 2.7$, assuming that the three-nearest neighbors are $\{cu_3, cu_4, cu_5\}$; the corresponding query $\sigma_{[opt\text{-}zoom=3x, lcd\text{-}size=2.7]}(\mathrm{P})$ would result in the empty set $\emptyset$. The handling of such situations will be discussed in Subsection 4.3.2.

Selecting the next question. Besides using defaults to support the user in the requirements specification process, the interaction log and the default mechanism can also be applied for identifying properties that may be interesting for the user within the scope of a recommendation session. For example, if a user has already specified requirements regarding the properties *price* and *opt-zoom*, defaults could propose properties that the user could be interested to specify next. Concepts supporting such a functionality are discussed in the following paragraphs.

Proposing defaults for properties to be presented next is an important functionality, as most users are not interested in specifying values for all properties – they rather want to specify the conditions that are important for them, but then immediately move on to see the recommended items. Different approaches to the selection of interesting questions are discussed by Mahmood and Ricci (2007). The precondition for such approaches is the availability of user interaction logs (see Table 4.4). One basic approach to the determination of defaults for the presentation of selectable customer properties is discussed by Mahmood and Ricci (2007), in which question recommendation is based on the principle of frequent usage (popularity). Such a popularity value can be calculated using Formula 4.5, in which the recommendation of a question depends strictly on the number of previous selections of other users – see, for example, Table 4.4. By analyzing the interaction log of Table 4.4, $popularity(price, pos:1) = 0.6$, whereas $popularity(mpix, pos:1) = 0.4$. Consequently, for the first question, the property *price* would be selected.

$$popularity(attribute, pos) = \frac{\#selections(attribute, pos)}{\#sessions} \tag{4.5}$$

Another approach for supporting question selection is to apply weighted-majority voters (Felfernig and Burke 2008). If, for example, a user has already

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.4. *Order of selected customer properties; for example, in session 4* (ID = 4) mpix *has been selected as first customer property to be specified.*

ID	pos:1	pos:2	pos:3	pos:4	pos:5	pos:6	···
1	price	opt-zoom	mpix	movies	LCD-size	sound	···
2	price	opt-zoom	mpix	movies	LCD-size	–	···
3	price	mpix	opt-zoom	lcd-size	movies	sound	···
4	mpix	price	opt-zoom	lcd-size	movies	–	···
5	mpix	price	lcd-size	opt-zoom	movies	sound	···

selected the properties *price* and *opt-zoom*, the weighted majority voter would identify the sessions with ID $\{1, 2\}$ as nearest neighbors (see Table 4.4) for the given set of requirements and then propose *mpix* as the next interesting question.

4.3.2 Dealing with unsatisfiable requirements and empty result sets

In our example, a given set of requirements $REQ = \{r_1 : price <= 150, r_2 : opt\text{-}zoom = 5x, r_3 : sound = yes, r_4 : waterproof = yes\}$ cannot be fulfilled by any of the products in $\mathrm{P} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ because $\sigma_{[price<=150, opt\text{-}zoom=5x, sound=yes, waterproof=yes]}(\mathrm{P}) = \emptyset$.

Many recommender systems are not able to propose a way out of such a "no solution could be found" dilemma. One option to help the user out is to incrementally and automatically relax constraints of the recommendation problem until a corresponding solution has been found. Different approaches to deal with this problem have been proposed in the literature. All of them share the same basic goal of identifying relaxations to the original set of constraints (Jannach 2006a, O'Sullivan et al. 2007, Felfernig et al. 2004, Felfernig et al. 2009). For the sake of better understandability, we assume that the user's requirements are directly related to item properties V_{PROD}.

In this section we discuss one basic approach in more detail. This approach is based on the idea of identifying and resolving requirements-immanent conflicts induced by the set of products in P. In such situations users ask for help that can be provided, for example, by the indication of a *minimal* set of requirements that should be changed in order to find a solution. In addition to a point to such unsatisfiable requirements, users could also be interested in repair proposals – that is, in adaptations of the initial requirements in such a way that the recommender is able to calculate a solution (Felfernig et al. 2009).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

The calculation of such repairs can be based on the concepts of model-based diagnosis (MBD; Reiter 1987) – the basis for the automated identification and repair of minimal sets of faulty requirements (Felfernig et al. 2004). MBD starts with a description of a system that is, in the case of recommender applications, a predefined set of products $p_i \in P$. If the actual system behavior is in contradiction to the intended system behavior (the unintended behavior is reflected by the fact that no solution could be found), the diagnosis task is to identify the system components (in our context represented by the user requirements in *REQ*) that, when we assume that they function abnormally, explain the discrepancy between the actual and the intended behavior of the system under consideration.

In the context of our problem setting, a diagnosis is a minimal set of user requirements whose repair (adaptation) will allow the retrieval of a recommendation. Given $P = \{p_1, p_2, \ldots, p_n\}$ and $REQ = \{r_1, r_2, \ldots, r_m\}$ where $\sigma_{[REQ]}(P) = \emptyset$, a knowledge-based recommender system would calculate a set of diagnoses $\triangle = \{d_1, d_2, \ldots, d_k\}$ where $\sigma_{[REQ-d_i]}(P) \neq \emptyset \; \forall d_i \in \triangle$. A diagnosis is a minimal set of elements $\{r_1, r_2, \ldots, r_k\} = d \subseteq REQ$ that have to be repaired in order to restore consistency with the given product assortment so at least one solution can be found: $\sigma_{[REQ-d]}(P) \neq \emptyset$. Following the basic principles of MBD, the calculation of diagnoses $d_i \in \triangle$ is based on the determination and resolution of conflict sets. A conflict set CS (Junker 2004) is defined as a subset $\{r_1, r_2, \ldots, r_l\} \subseteq REQ$, such that $\sigma_{[CS]}(\mathrm{P}) = \emptyset$. A conflict set CS is minimal if and only if (iff) there does not exist a CS' with $CS' \subset CS$.

As mentioned, no item in P completely fulfills the requirements $REQ = \{r_1 : price <= 150, r_2 : opt\text{-}zoom = 5\times, r_3 : sound{=}yes, r_4 : waterproof{=}yes\}$: $\sigma_{[price<=150, opt\text{-}zoom=5x, sound=yes, waterproof=yes]}(P) = \emptyset$. The corresponding conflict sets are $CS_1 = \{r_1, r_2\}$, $CS_2 = \{r_2, r_4\}$, and $CS_3 = \{r_1, r_3\}$, as $\sigma_{[CS1]}(P) = \emptyset$, $\sigma_{[CS2]}(P) = \emptyset$, and $\sigma_{[CS3]}(P) = \emptyset$. The identified conflict sets are minimal, as $\neg\exists CS_1'$: $CS_1' \subset CS_1 \wedge \sigma_{[CS1']}(\mathrm{P}) = \emptyset$, $\neg\exists CS_2'$: $CS_2' \subset CS_2 \wedge \sigma_{[CS2']}(\mathrm{P}) = \emptyset$, and $\neg\exists CS_3'$: $CS_3' \subset CS_3 \wedge \sigma_{[CS3']}(\mathrm{P}) = \emptyset$.

Diagnoses $d_i \in \triangle$ can be calculated by resolving conflicts in the given set of requirements. Because of its minimality, one conflict can be easily resolved by deleting one of the elements from the conflict set. After having deleted at least one element from each of the identified conflict sets, we are able to present a corresponding diagnosis. The diagnoses derived from the conflict sets $\{CS_1, CS_2, CS_3\}$ in our working example are $\triangle = \{d_1{:}\{r_1, r_2\}, d_2{:}\{r_1, r_4\}, d_3{:}\{r_2, r_3\}\}$. The calculation of such diagnoses (see Figure 4.1) starts with the first identified conflict set (CS_1) (1). CS_1 can be resolved in two alternative ways: by deleting either r_1 or r_2. Both of these alternatives are explored following a breadth-first search regime. After deleting r_1 from *REQ*, the next conflict

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

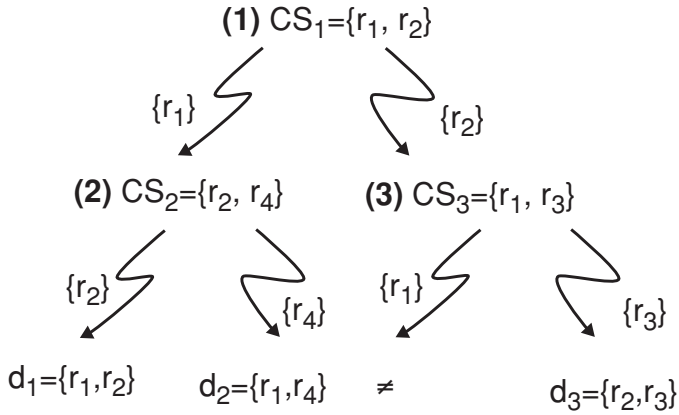

Figure 4.1. Calculating diagnoses for unsatisfiable requirements.

set is CS_2 (2), which also allows two different relaxations, namely r_2 and r_4. Deleting the elements of CS_2 leads to the diagnoses d_1 and d_2. After deleting r_2 from CS_1, the next returned conflict set is CS_3 (3). Both alternative deletions for CS_3, in principle, lead to a diagnosis. However, the diagnosis $\{r_1, r_2\}$ is already contained in d_1; consequently, this path is not expanded further, and the third and final diagnosis is d_3.

Calculating conflict sets. A recent and general method for the calculation of conflict sets is QUICKXPLAIN (Algorithm 4.1), an algorithm that calculates one conflict set at a time for a given set of constraints. Its divide-and-conquer strategy helps to significantly accelerate the performance compared to other approaches (for details see, e.g., Junker 2004).

QUICKXPLAIN has two input parameters: first, P is the given product assortment $P = \{p_1, p_2, \ldots, p_m\}$. Second, $REQ = \{r_1, r_2, \ldots, r_n\}$ is a set of requirements analyzed by the conflict detection algorithm.

QUICKXPLAIN is based on a recursive divide-and-conquer strategy that divides the set of requirements into the subsets REQ_1 and REQ_2. If both subsets contain about 50 percent of the requirements (the splitting factor is $\frac{n}{2}$), all the requirements contained in REQ_2 can be deleted (ignored) after a single consistency check if $\sigma_{[REQ1]}(P) = \emptyset$. The splitting factor of $\frac{n}{2}$ is generally recommended; however, other factors can be defined. In the best case (e.g., all elements of the conflict belong to subset REQ_1) the algorithm requires $\log_2 \frac{n}{u} + 2u$ consistency checks; in the worst case, the number of consistency checks is $2u(\log_2 \frac{n}{u} + 1)$, where u is the number of elements contained in the conflict set.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Algorithm 4.1 QUICKXPLAIN(P, REQ)

Input: trusted knowledge (items) P; Set of requirements REQ
Output: minimal conflict set CS
if $\sigma_{[REQ]}(P) \neq \emptyset$ or $REQ = \emptyset$ **then** return $\emptyset$
else return QX′ (P, $\emptyset$, $\emptyset$, REQ);

Function QX′(P, B, Δ, REQ)
if $\Delta \neq \emptyset$ and $\sigma_{[B]}(P) = \emptyset$ **then** return $\emptyset$;
if $REQ = \{r\}$ **then** return $\{r\}$;
let $\{r_1, \ldots, r_n\} = REQ$;
let k be $\frac{n}{2}$;
$REQ_1 \leftarrow r_1, \ldots, r_k$ and $REQ_2 \leftarrow r_{k+1}, \ldots, r_n$;
$\Delta_2 \leftarrow$ QX′(P, $B \cup REQ_1$, REQ_1, REQ_2);
$\Delta_1 \leftarrow$ QX′(P, $B \cup \Delta_2$, Δ_2, REQ_1);
return $\Delta_1 \cup \Delta_2$;

To show how the algorithm QUICKXPLAIN works, we will exemplify the calculation of a conflict set on the basis of our working example (see Figure 4.2) – that is, P = $\{p_1, p_2, \ldots, p_8\}$ and REQ = $\{r_1$:price≤150, r_2:opt-zoom=5x, r_3:sound=yes, r_4:waterproof=yes$\}$. First, the main routine is activated (1), which checks whether $\sigma_{[REQ]}(P) \neq \emptyset$. As this is not the case, the recursive routine QX′ is activated (2). This call results in call (3) (to obtain Δ_2), which itself results in $\emptyset$, as $\Delta \neq \emptyset$ and $\sigma_{[B]}(P) = \emptyset$. To obtain Δ_1, call (4) directly activates call (5) and call (6), and each those last calls identifies a corresponding conflict element (r_2 and r_1). Thus, CS_1:$\{r_1, r_2\}$ is returned as the first conflict set.

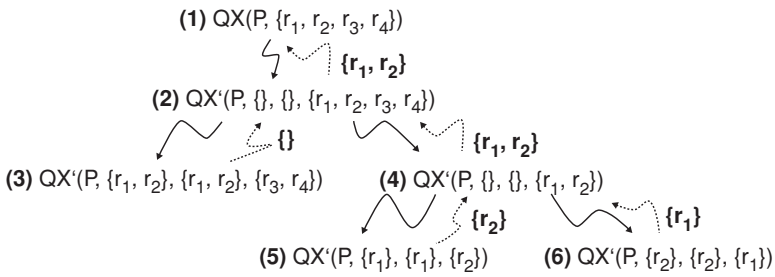

Figure 4.2. Example: calculation of conflict sets using QUICKXPLAIN.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Algorithm 4.2 MinRelax(P, REQ)

Input: Product assortment P; set of requirements REQ
Output: Complete set of all minimal diagnoses Δ
$\Delta \leftarrow \emptyset$;
forall $p_i \in$ P **do**
  PSX $\leftarrow$ product-specific-relaxation(p_i, REQ);
  SUB $\leftarrow \{r \in \Delta - r \subset PSX\}$;
  if SUB $\neq \emptyset$ **then** continue with next p_i;
  SUPER $\leftarrow \{r \in \Delta - PSX \subset r\}$;
  if SUPER $\neq \emptyset$ **then** $\Delta \leftarrow \Delta -$ SUPER;
  $\Delta \leftarrow \Delta \cup \{PSX\}$;
return Δ;

Besides the usage within an MBD procedure, the conflicts computed with QuickXPlain can also be used in interactive relaxation scenarios as described by McSherry (2004), in which the user is presented with one or more remaining conflicts and asked to choose one of the conflict elements to retract. For an example of such an algorithm, see Jannach 2006b.

Fast in-memory computation of relaxations with MinRelax. As long as the set of items is specified explicitly (as in Table 4.1), the calculation of diagnoses can be achieved without the explicit determination and resolution of conflict sets (Jannach 2006a). MinRelax (Algorithm 4.2) is such an algorithm to determine the complete set of diagnoses. The previously discussed approach based on the resolution of conflict sets is still indispensable in interactive settings, in which users should be able to manually resolve conflicts, and in settings in which items are not enumerated but described in the form of generic product structures (Felfernig et al. 2004).

The MinRelax algorithm for determining the complete set of minimal diagnoses has been introduced by Jannach (2006a). This algorithm calculates, for each item $p_i \in P$ and the requirements in REQ, a corresponding product-specific relaxation PSX. PSX is a minimal diagnosis $d \in \Delta$ (the set of all minimal diagnoses) if there is no set r such that $r \subset PSX$. For example, the PSX for item p_1 and the requirements $\{r_1, r_2, r_3, r_4\}$ is the ordered set $\{1, 0, 0, 1\}$, which corresponds to the first column of Table 4.5 (only the requirements r_1 and r_4 are satisfied by item p_1).

The performance of MinRelax is $(n * (n + 1))/2$ subset checks in the worst case, which can be conducted efficiently with in-memory bitset

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.5. *Intermediate representation: item-specific relaxations PSX for* $p_i \in P$.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
r_1 : *price* ≤ 150	1	0	0	0	0	0	0	0
r_2 : *opt-zoom* $= 5\times$	0	1	0	0	0	0	0	0
r_3 : *sound* $=$ *yes*	0	1	1	0	1	1	1	1
r_4 : *waterproof* $=$ *yes*	1	0	0	1	0	0	0	1

operations (Jannach 2006a). Table 4.5 depicts the relationship between our example requirements and each $p_i \in P$. Because of the explicit enumeration of all possible items in P, we can determine for each requirement/item combination whether the requirement is supported by the corresponding item. Each column of Table 4.5 represents a diagnosis; our goal is to identify the diagnoses that are minimal.

4.3.3 Proposing repairs for unsatisfiable requirements

After having identified the set of possible diagnoses ($\triangle$), we must propose repair actions for each of those diagnoses – in other words, we must identify possible adaptations for the existing set of requirements such that the user is able to find a solution (Felfernig et al. 2009). Alternative repair actions can be derived by querying the product table P with $\pi_{[attributes(d)]}\sigma_{[REQ-d]}(P)$. This query identifies all possible repair alternatives for a single diagnosis d $\in \Delta$ where $\pi_{[attributes(d)]}$ is a projection and $\sigma_{[REQ-d]}(P)$ is a selection of -tuples from P that satisfy the criteria in *REQ–d*. Executing this query for each of the identified diagnoses produces a complete set of possible repair alternatives. For reasons of simplicity we restrict our example to three different repair alternatives, each belonging to exactly one diagnosis. Table 4.6 depicts the complete set of repair alternatives $REP = \{rep_1, rep_2, rep_3\}$ for our working example, where

- $\pi_{[attributes(d1)]}\sigma_{[REQ-d1]}(P) = \pi_{[price,opt\text{-}zoom]}\sigma_{[r3:sound=yes,r4:waterproof=yes]}(P) = \{price{=}278, opt\text{-}zoom{=}10\times\}$
- $\pi_{[attributes(d2)]}\sigma_{[REQ-d2]}(P) = \pi_{[price,waterproof]}\sigma_{[r2:opt\text{-}zoom=5x,r3:sound=yes]}(P) = \{price{=}182, waterproof{=}no\}$
- $\pi_{[attributes(d3)]}\sigma_{[REQ-d3]}(P) = \pi_{[opt\text{-}zoom,sound]}\sigma_{[r1:price<=150,r4:waterproof=yes]}(P) = \{opt\text{-}zoom{=}4\times, sound{=}no\}$

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.6. *Repair alternatives for requirements in REQ.*

repair	price	opt-zoom	sound	waterproof
rep_1	278	10×	√	√
rep_2	182	√	√	no
rep_3	√	4×	no	√

4.3.4 Ranking the items/utility-based recommendation

It is important to rank recommended items according to their utility for the customer. Because of primacy effects that induce customers to preferably look at and select items at the beginning of a list, such rankings can significantly increase the trust in the recommender application as well as the willingness to buy (Chen and Pu 2005, Felfernig et al. 2007)

In knowledge-based conversational recommenders, the ranking of items can be based on the multi-attribute utility theory (MAUT), which evaluates each item with regard to its utility for the customer. Each item is evaluated according to a predefined set of dimensions that provide an aggregated view on the basic item properties. For example, *quality* and *economy* are dimensions in the domain of digital cameras; *availability*, *risk*, and *profit* are such dimensions in the financial services domain. Table 4.7 exemplifies the definition of scoring rules that define the relationship between item properties and dimensions. For example, a digital camera with a *price* lower than or equal to 250 is evaluated, with Q score of 5 regarding the dimension *quality* and 10 regarding the dimension *economy*.

We can determine the utility of each item p in P for a specific customer (Table 4.8). The customer-specific item utility is calculated on the basis of Formula 4.6, in which the index j iterates over the number of predefined dimensions (in our example, #(dimensions)=2: *quality* and *economy*), *interest*(j) denotes a user's interest in dimension j, and *contribution*(p, j) denotes the contribution of item p to the interest dimension j. The value for *contribution*(p, j) can be calculated by the scoring rules defined in Table 4.7 – for example, the contribution of item p_1 to the dimension *quality* is $5 + 4 + 6 + 6 + 3 + 7 + 10 = 41$, whereas its contribution to the dimension *economy* is $10 + 10 + 9 + 10 + 10 + 10 + 6 = 65$.

To determine the overall utility of item p_1 for a specific customer, we must take into account the customer-specific interest in each of the given dimensions – *interest*(j). For our example we assume the customer preferences depicted in Table 4.9. Following Formula 4.6, for customer cu_1, the

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.7. *Example scoring rules regarding the dimensions* quality *and* economy.

	value	quality	economy
price	≤250	5	10
	>250	10	5
mpix	≤8	4	10
	>8	10	6
opt-zoom	≤9	6	9
	>9	10	6
LCD-size	≤2.7	6	10
	>2.7	9	5
movies	yes	10	7
	no	3	10
sound	yes	10	8
	no	7	10
waterproof	yes	10	6
	no	8	10

utility of item p_2 is 49*0.8 + 64*0.2 = 52.0 and the overall utility of item p_8 would be 69*0.8 + 43*0.2 = 63.8. For customer cu_2, item p_2 has the utility 49*0.4 + 64*0.6 = 58.0 and item p_8 has the utility 69*0.4 + 43*0.6 = 53.4. Consequently, item p_8 has a higher utility (and the highest utility) for cu_1, whereas item p_2 has a higher utility (and the highest utility) for cu_2. Formula 4.6 follows the principle of the similarity metrics introduced in Section 4.2: *interest*(j) corresponds to the weighting of requirement r_j and *contribution*(p, j) corresponds to the local similarity function *sim*(p, r) (McSherry 2003a).

Table 4.8. *Item utilities for customer cu_1 and customer cu_2.*

	quality	economy	cu_1	cu_2
p_1	$\sum(5,4,6,6,3,7,10) = 41$	$\sum(10,10,9,10,10,10,6) = 65$	45.8 [8]	55.4 [6]
p_2	$\sum(5,4,6,6,10,10,8) = 49$	$\sum(10,10,9,10,7,8,10) = 64$	52.0 [7]	58.0 [1]
p_3	$\sum(5,4,10,6,10,10,8) = 53$	$\sum(10,10,6,10,7,8,10) = 61$	54.6 [5]	57.8 [2]
p_4	$\sum(5,10,10,6,10,7,10) = 58$	$\sum(10,6,6,10,7,10,6) = 55$	57.4 [4]	56.2 [4]
p_5	$\sum(5,4,6,10,10,10,8) = 53$	$\sum(10,10,9,6,7,8,10) = 60$	54.4 [6]	57.2 [3]
p_6	$\sum(5,10,6,9,10,10,8) = 58$	$\sum(10,6,9,5,7,8,10) = 55$	57.4 [3]	56.2 [5]
p_7	$\sum(10,10,6,9,10,10,8) = 63$	$\sum(5,6,9,5,7,8,10) = 50$	60.4 [2]	55.2 [7]
p_8	$\sum(10,10,10,9,10,10,10) = 69$	$\sum(5,6,6,5,7,8,6) = 43$	63.8 [1]	53.4 [8]

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.9. *Customer-specific preferences represent the values for* interest(j) *in Formula 4.6.*

customer (user)	quality	economy
cu_1	80%	20%
cu_2	40%	60%

The concepts discussed here support the calculation of personalized rankings for a given set of items. However, such utility-based approaches can be applied in other contexts as well – for example, the calculation of utilities of specific repair alternatives (personalized repairs; Felfernig et al. 2006) or the calculation of utilities of explanations (Felfernig et al. 2008b).

$$utility(p) = \sum_{j=1}^{\#(dimensions)} interest(j) * contribution(p, j) \qquad (4.6)$$

There exist different approaches to determining a customer's degree of interest in a certain dimension (*interest*(*j*) in Formula 4.6). Such preferences can be explicitly defined by the user (user-defined preferences). Preferences can also be predefined in the form of scoring rules (utility-based preferences) derived by analyzing logs of previous user interactions (e.g., conjoint analysis). These basic approaches will be exemplified in the following paragraphs.

User-defined preferences. The first and most straightforward approach is to directly ask the customer for his or her preferences within the scope of a recommendation session. Clearly, this approach has the main disadvantage that the overall interaction effort for the user is nearly doubled, as for many of the customer properties the corresponding importance values must be specified. A second problem with this basic approach is that the recommender user interface is obtrusive in the sense that customers are interrupted in their preference construction process and are forced to explicitly specify their preferences beforehand.

Utility-based preferences. A second possible approach to determining customer preferences is to apply the scoring rules of Table 4.7. If we assume, for example, that a customer has specified the requirements $REQ = \{r_1 : price <= 200, r_2 : mpix = 8.0, r_3 : opt\text{-}zoom = 10\times, r_4 : lcd\text{-}size <= 2.7\}$, we can directly derive the instantiations of the corresponding dimensions by applying the scoring rules in Table 4.7. In our case, the dimension *quality*

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.10. *Ranking of price/mpix stimuli: $price_1$[100–159], $price_2$[160–199], $price_3$[200–300], $mpix_1$[5.0–8.0], and $mpix_2$[8.1–11.0].*

	$price_1$	$price_2$	$price_3$	$avg(mpix_x)$
$mpix_1$	4	5	6	5
$mpix_2$	2	1	3	2
$avg(price_x)$	3	3	4.5	3.5

would be $5 + 4 + 10 + 6 = 25$ and the dimension *economy* would be $10 + 10 + 6 + 10 = 36$. This would result in a relative importance for *quality* with a value of $\frac{25}{25+36} = 0.41$ and a relative importance for *economy* with the value $\frac{36}{25+36} = 0.59$.

Conjoint analysis. The following simple example should characterize the basic principle of conjoint analysis (Belanger 2005). In this example, a user (test person) is confronted with different *price/mpix* value combinations (stimuli). The user's task is to rank those combinations; for example, the combination $mpix_2$[8.1–11.0] / $price_2$[160–199] gets the highest ranking (see Tables 4.10 and 4.11). The average values for the columns inform us about the average ranking for the corresponding *price* interval ($avg(price_x)$). The average values for the rows inform us about the average rankings for the corresponding *mpix* interval $avg(mpix_x)$. The average value over all rankings is $avg(ranking) = 3.5$.

The information we can extract from Tables 4.10 and 4.11 is the deviation from the average ranking for specific property values – for

Table 4.11. *Effects of customer property changes on overall utility: changes in* mpix *have a higher impact on the overall utility than changes in price.*

	$avg(ranking) - avg(mpix_x)$
$avg(ranking) - avg(mpix_1)$	−1.5
$avg(ranking) - avg(mpix_2)$	1.5
$avg(ranking) - avg(price_1)$	0.5
$avg(ranking) - avg(price_2)$	0.5
$avg(ranking) - avg(price_3)$	−1.0

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

example, $avg(price_x)$ from $avg(ranking)$: $avg(ranking) - avg(price_1) = 0.5$, $avg(ranking) - avg(price_2) = 0.5$, $avg(ranking) - avg(price_3) = -1$. Furthermore, $avg(ranking) - avg(mpix_1) = -1.5$ and $avg(ranking) - avg(mpix_2) = 1.5$. The $avg(price_x)$ span is 1.5 ($-1 \ldots 0.5$) whereas the span for $avg(mpix_x)$ is 3.0 ($-1.5 \ldots 1.5$). Following the ideas of conjoint analysis (Belanger 2005) we are able to conclude that *price* changes have a lower effect on the overall utility (for this customer) than changes in terms of *megapixels*. This result is consistent with the idea behind the ranking in our example (Tables 4.10 and 4.11), as the highest ranking was given to the combination $price_2/mpix_2$, where a higher price was accepted in order to ensure high quality of technical features (here: *mpix*). Consequently, we can assign a higher importance to the technical property *mpix* compared with the property *price*.

In this section we provided an overview of concepts that typically support users in the interaction with a constraint-based recommender application. *Diagnosis and repair* concepts support users in situations in which no solution could be found. *Defaults* provide support in the requirements specification process by proposing reasonable alternatives – a negative connotation is that defaults can be abused to manipulate users. *Utility-based ranking* mechanisms support the ordering of information units such as items on a result page, repair alternatives provided by a diagnosis and repair component, and the ranking of explanations for recommended items. These concepts form a toolset useful for the implementation of constraint-based recommender applications. A commercial application built on the basis of those concepts is presented in Section 4.5.

4.4 Interacting with case-based recommenders

Similar to constraint-based recommenders, earlier versions of case-based recommenders followed a pure *query-based approach*, in which users had to specify (and often respecify) their requirements until a target item (an item that fits the user's wishes and needs) has been identified (Burke 2002a). Especially for nonexperts in the product domain, this type of requirement elicitation process can lead to tedious recommendation sessions, as the interdependent properties of items require a substantial domain knowledge to perform well (Burke 2002a). This drawback of pure query-based approaches motivated the development of *browsing-based approaches* to item retrieval, in which users – maybe not knowing what they are seeking – are navigating in the item space with the goal to find useful alternatives. *Critiquing* is an effective way to

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

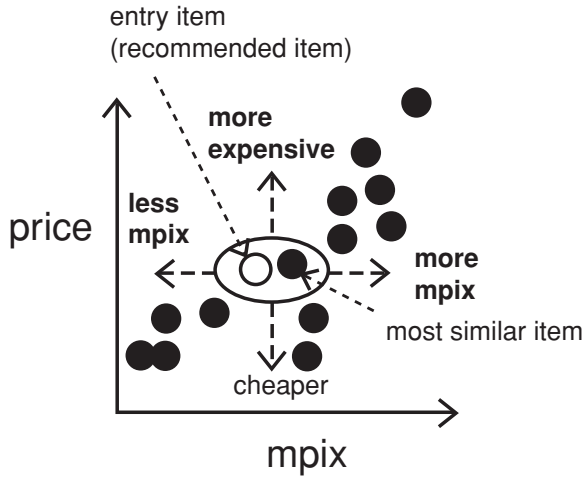

Figure 4.3. Critique-based navigation: items recommended to the user can be critiqued regarding different item properties (e.g., *price* or *mpix*).

support such navigations and, in the meantime, it is one of the key concepts of case-based recommendation; this concept will be discussed in detail in the following subsections.

4.4.1 Critiquing

The idea of critiquing (Burke 2000, Burke et al. 1997) is that users specify their change requests in the form of goals that are not satisfied by the item currently under consideration (*entry item* or *recommended item*). If, for example, the *price* of the currently displayed digital camera is too high, a critique *cheaper* can be activated; if the user wants to have a camera with a higher resolution (*mpix*), a corresponding critique *more mpix* can be selected (see Figure 4.3).

Further examples for critiques are "the hotel location should be nearer to the sea" or "the apartment should be more modern-looking". Thus, critiques can be specified on the level of technical properties as well as on the level of abstract dimensions.

State-of-the-art case-based recommenders are integrating query-based with browsing-based item retrieval (Burke 2002a). On one hand, critiquing supports an effective navigation in the item space; on the other hand, similarity-based case retrieval supports the identification of the *most similar items* – that is, items similar to those currently under consideration. Critiquing-based recommender

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Algorithm 4.3 SIMPLECRITIQUING(q, CI)

```
Input: Initial user query q; Candidate items CI
procedure SimpleCritiquing(q, CI)
 repeat
  r ← ItemRecommend(q, CI);
  q ← UserReview(r, CI);
 until empty(q)
end procedure

procedure ItemRecommend(q, CI)
 CI ← {ci ∈ CI: satisfies(ci, q)};
 r ← mostsimilar(CI, q);
 return r;
end procedure

procedure UserReview(r, CI)
 q ← critique(r);
 CI ← CI − r;
 return q;
end procedure
```

systems allow users to easily articulate preferences without being forced to specify concrete values for item properties (see the previous example). The goal of critiquing is to achieve time savings in the item selection process and, at the same time, achieve at least the same recommendation quality as standard query-based approaches. The major steps of a critiquing-based recommender application are the following (see Algorithm 4.3, SIMPLECRITIQUING).

Item recommendation. The inputs for the algorithm SIMPLECRITIQUING[4] are an initial *user query* q, which specifies an initial set of requirements, and a set of *candidate items* CI that initially consists of all the available items (the product assortment). The algorithm first activates the procedure ITEMRECOMMEND, which is responsible for selecting an item r to be presented to the user. We denote the item that is displayed in the first critiquing cycle as *entry item* and all other items displayed thereafter as *recommended items*. In the first critiquing

[4] The notation used in the algorithm is geared to Reilly et al. (2005a).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

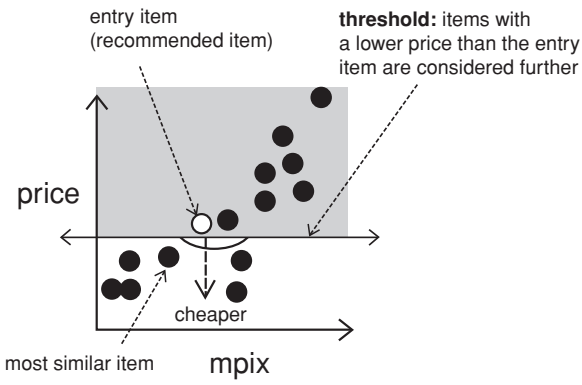

Figure 4.4. Critique-based navigation: remaining candidate items (items with bright background) after a critique on the entry item property *price*.

cycle, the retrieval of such items is based on a user query q that represents a set of initial requirements. Entry items are typically determined by calculating the similarity between the requirements and the candidate items. After the first critiquing cycle has been completed, recommended items are determined by the procedure ItemRecommend on the basis of the similarity between the currently recommended item and those items that fulfill the criteria of the critique specified by the user.

Item reviewing. The user reviews the recommended (entry) item and either accepts the recommendation or selects another critique, which triggers a new critiquing cycle (procedure UserReview). If a critique has been triggered, only the items (the *candidate items*) that fulfill the criteria defined in the critique are further taken into account – this reduction of CI is done in procedure ItemRecommend. For example, if a user activates the critique *cheaper*, and the *price* of the recommended (entry) camera is 300, the recommender excludes cameras with a *price* greater than or equal to 300 in the following critiquing cycle.

4.4.2 Compound critiquing

In our examples so far we primarily considered the concept of *unit critiques*; such critiques allow the definition of change requests that are related to a single item property. Unit critiques have a limited capability to effectively narrow down the search space. For example, the unit critique on *price* in Figure 4.4 eliminates only about half the items.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

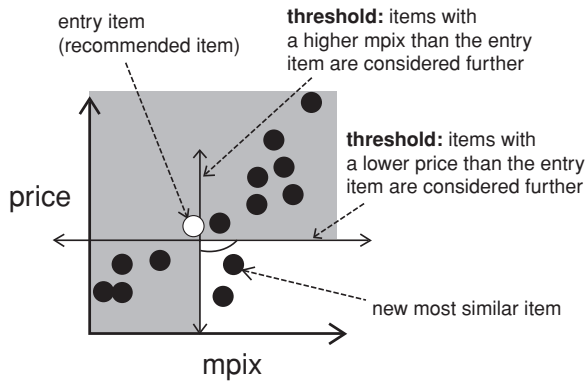

Figure 4.5. Critique-based navigation: remaining candidate items (items with bright background) after a compound critique on *price* and *mpix*.

Allowing the specification of critiques that operate over multiple properties can significantly improve the efficiency of recommendation dialogs, for example, in terms of a reduced number of critiquing cycles. Such critiques are denoted as *compound critiques*. The effect of compound critiques on the number of eliminated items (items not fulfilling the criteria of the critique) is shown in Figure 4.5. The compound critique *cheaper and more mpix* defines additional goals on two properties that should be fulfilled by the next proposed recommendation.

An important advantage of compound critiques is that they allow a faster progression through the item space. However, compound critiques still have disadvantages as long as they are formulated *statically*, as all critique alternatives are available for every item displayed. For example, in the context of a high-end computer with the *fastest CPU available* on the market and the *maximum available storage capacity*, a corresponding critique *faster CPU and more storage capacity* (or *more efficient*) would be still proposed by a static compound critiquing approach. In the following subsection we will present the *dynamic critiquing* approach that helps to solve this problem.

4.4.3 Dynamic critiquing

Dynamic critiquing exploits *patterns*, which are generic descriptions of differences between the recommended (entry) item and the candidate items – these patterns are used for the derivation of compound critiques. Critiques are denoted as dynamic because they are derived on the fly in each critiquing cycle. Dynamic critiques (Reilly et al. 2007b) are calculated using the concept

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Algorithm 4.4 DynamicCritiquing(q, CI)

Input: Initial user query q; Candidate items CI;
number of compound critiques per cycle k;
minimum support for identified association rules σ_{min}

```
procedure DynamicCritiquing(q, CI, k, σ_min)
 repeat
  r ← ItemRecommend(q, CI);
  CC ← CompoundCritiques(r, CI, k, σ_min);
  q ← UserReview(r, CI, CC);
 until empty(q)
end procedure

procedure ItemRecommend(q, CI)
 CI ← {ci ∈ CI: satisfies(ci, q)};
 r ← mostsimilar(CI, q);
 return r;
end procedure

procedure UserReview(r, CI, CC)
 q ← critique(r, CC);
 CI ← CI − r;
 return q;
end procedure

procedure CompoundCritiques(r, CI, k, σ_min)
 CP ← CritiquePatterns(r, CI);
 CC ← Apriori(CP, σ_min);
 SC ← SelectCritiques(CC, k);
 return SC;
end procedure
```

of association rule mining (Agrawal and Srikant 1994). Such a rule can be, for example, "42.9% of the remaining digital cameras have a higher zoom and a lower price". The critique that corresponds to this property combination is "more zoom and lower price". A dynamic critiquing cycle consists of the following basic steps (see Algorithm 4.4, DynamicCritiquing[5]).

[5] The algorithm has been developed by Reilly et al. (2005a).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

The inputs for the algorithm are an initial user query q, which specifies the initial set of requirements, a set of candidate items CI that initially consists of all the available items, k as the maximum number of compound critiques to be shown to the user in one critiquing cycle, and σ_{min} as the minimum support value for calculated association rules.

Item recommendation. Similar to the SIMPLECRITIQUING algorithm discussed in Section 4.4.1, the DYNAMICCRITIQUING algorithm first activates the procedure ITEMRECOMMEND, which is responsible for returning one *recommended item r* (respectively, *entry item* in the first critiquing cycle). On the basis of this item, the algorithm starts the calculation of compound critiques $cc_i \in CC$ by activating the procedure COMPOUNDCRITIQUES, which itself activates the procedures CRITIQUEPATTERNS (identification of critique patterns), APRIORI (mining compound critiques from critique patterns), and SELECTCRITIQUES (ranking compound critiques). These functionalities will be discussed and exemplified in the following paragraphs. The identified compound critiques in CC are then shown to the user in USERREVIEW. If the user selects a critique – which could be a unit critique on a specific item property as well as a compound critique – this forms the criterion of the new user query q. If the resulting query q is empty, the critiquing cycle can be stopped.

Identification of critique patterns. Critique patterns are a generic representation of the differences between the currently recommended item (entry item) and the candidate items. Table 4.12 depicts a simple example for the derivation of critique patterns, where item ei_8 is assumed to be the entry item and the items $\{ci_1, \ldots, ci_7\}$ are the candidate items. On the basis of this example, critique patterns can be easily generated by comparing the properties of item ei_8 with the properties of $\{ci_1, \ldots, ci_7\}$. For example, compared with item ei_8, item ci_1 is cheaper, has less *mpix*, a lower *opt-zoom*, a smaller *lcd-size*, and does not have a *movie* functionality. The corresponding critique pattern for item ci_1 is $(<, <, <, <, \neq)$. A complete set of critiquing patterns in our example setting is shown in Table 4.12. These patterns are the result of calculating the type of difference for each combination of *recommended* (*entry*) and *candidate item*. In the algorithm DYNAMICCRITIQUING, critique patterns are determined on the basis of the procedure CRITIQUINGPATTERNS.

Mining compound critiques from critique patterns. The next step is to identify compound critiques that frequently co-occur in the set of critique patterns. This approach is based on the assumption that critiques correspond to feature combinations of interest to the user – that is, a user would like to adapt the

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.12. *Critique patterns (CP) are generated by analyzing the differences between the* recommended item *(the* entry item*, EI) and the* candidate items *(CI). In this example, item ei_8 is assumed to be the* entry item*, $\{ci_1, \ldots, ci_7\}$ are assumed to be the candidate items, and $\{cp_1, \ldots, cp_7\}$ are the critique patterns.*

	id	price	mpix	opt-zoom	LCD-size	movies
entry item (EI)	ei_8	278	9.1	9×	3.0	yes
	ci_1	148	8.0	4×	2.5	no
	ci_2	182	8.0	5×	2.7	yes
	ci_3	189	8.0	10×	2.5	yes
candidate items (CI)	ci_4	196	10.0	12×	2.7	yes
	ci_5	151	7.1	3×	3.0	yes
	ci_6	199	9.0	3×	3.0	yes
	ci_7	259	10.0	10×	3.0	yes
	cp_1	$<$	$<$	$<$	$<$	$\neq$
	cp_2	$<$	$<$	$<$	$<$	$=$
	cp_3	$<$	$<$	$>$	$<$	$=$
critique patterns (CP)	cp_4	$<$	$>$	$>$	$<$	$=$
	cp_5	$<$	$<$	$<$	$=$	$=$
	cp_6	$<$	$<$	$<$	$=$	$=$
	cp_7	$<$	$>$	$>$	$=$	$=$

requirements in exactly the proposed combination (Reilly et al. 2007b). For critique calculation, Reilly et al. (2007b) propose applying the APRIORI algorithm (Agrawal and Srikant 1994). The output of this algorithm is a set of *association rules* $p \Rightarrow q$, which describe relationships between elements in the set of critique patterns. An example is $>_{zoom} \Rightarrow <_{price}$, which can be derived from the critique patterns of Table 4.12. This rule denotes the fact that given $>_{zoom}$ as part of a critique pattern, $<_{price}$ is contained in the same critique pattern. Examples for association rules and the corresponding compound critiques that can be derived from the critique patterns in Table 4.12 are depicted in Table 4.13.

Each association rule is additionally characterized by *support* and *confidence* values. Support (SUPP) denotes the number of critique patterns that include all the elements of the antecedent and consequent of the association rule (expressed in terms of the percentage of the number of critique patterns). For example, the support of association rule ar_1 in Table 4.13 is 28.6 percent; of the seven critique patterns, exactly two include the antecedent and consequent part of association rule ar_1. Confidence (CONF) denotes the ratio between critique

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.13. *Example association rules (AR) and the compound critiques (CC) derived from CP in Table 4.12.*

association rules (AR)	compound critiques (CC)	SUPP	CONF
ar_1: $>_{mpix} \Rightarrow >_{zoom}$	cc_1: $>_{mpix(9.1)}$, $>_{zoom(9x)}$	28.6	100.0
ar_2: $>_{zoom} \Rightarrow <_{price}$	cc_2: $>_{zoom(9x)}$, $<_{price(278)}$	42.9	100.0
ar_3: $=_{movies} \Rightarrow <_{price}$	cc_3: $=_{movie(yes)}$, $<_{price(278)}$	85.7	100.0

patterns containing all the elements of the antecedent and consequent of the association rule and those containing only the antecedent part. For all the association rules in Table 4.13 the confidence level is 100.0 percent – that is, if the antecedent part of the association rule is confirmed by the pattern, the consequent part is confirmed as well. In the algorithm DynamicCritiquing, compound critiques are determined on the basis of the procedure APriori that represents a basic implementation of the Apriori algorithm (Agrawal and Srikant 1994).

Ranking of compound critiques. The number of compound critiques can become very large, which makes it important to filter out the most relevant critiques for the user in each critiquing cycle. Critiques with *low support* have the advantage of significantly reducing the set of candidate items, but at the same time they decrease the probability of identifying the target item. Critiques with *high support* can significantly increase the probability of finding the target item. However, these critiques eliminate a low number of candidate cases, which leads to a larger number of critiquing cycles in recommendation sessions. Many existing recommendation approaches rank compound critiques according to the support values of association rules, because the lower the support of the corresponding association rules, the more candidate items can be eliminated. In our working example, such a ranking of compound critiques $\{cc_1, cc_2, cc_3\}$ is cc_1, cc_2, and cc_3. Alternative approaches to the ranking of compound critiques are discussed, for example, by Reilly et al. (2004), where low support, high support, and random critique selection have been compared. This study reports a lower number of interaction cycles in the case that compound critiques are sorted ascending based on their support value. The issue of critique selection is in need of additional empirical studies focusing on the optimal balance between a low number of interaction cycles and the number of excluded candidate items. In the algorithm DynamicCritiquing, compound critiques are selected on the basis of the procedure SelectCritiques.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Item reviewing. At this stage of a recommendation cycle all the relevant information for deciding about the next action is available for the user: the recommended (entry) item and the corresponding set of compound critiques. The user reviews the recommended item and either accepts the recommendation or selects a critique (unit or compound), in which case a new critiquing cycle is started. In the algorithm DynamicCritiquing, item reviews are conducted by the user in UserReview.

4.4.4 Advanced item recommendation

After a critique has been selected by the user, the next item must be proposed (*recommended item* for the next critiquing cycle). An approach to doing this – besides the application of simple similarity measures (see Section 4.2) – is described by Reilly et al. (2007b), where a *compatibility score* is introduced that represents the percentage of compound critiques $cc_i \in CC_U$ that already have been selected by the user and are consistent with the candidate item ci. This compatibility-based approach to item selection is implemented in Formula 4.7.

$$compatibility(ci, CC_U) = \frac{|\{cc_i \in CC_U : satisfies(cc_i, ci)\}|}{|CC_U|} \tag{4.7}$$

CC_U represents a set of (compound) critiques already selected by the user; $satisfies(cc_i, ci) = 1$ indicates that critique cc_i is *consistent* with candidate item ci and $satisfies(cc_i, ci) = 0$ indicates that critique cc_i is *inconsistent* with ci. On the basis of this compatibility measure, Reilly et al. (2007b) introduce a new quality measure for a certain candidate item ci (see Formula 4.8). This formula assigns the highest values to candidate items ci that are as compatible as possible with the already selected compound critiques and also as similar as possible to the currently recommended item ri.

$$quality(ci, ri, CC_U) = compatibility(ci, CC_U) * similarity(ci, ri) \tag{4.8}$$

Table 4.14 exemplifies the application of Formula 4.8. Let us assume that $CC_U = \{cc_2\text{: } >_{zoom(9x)}, <_{price(278)}\}$ is the set of critiques that have been selected by the user in UserReview – in other words, only one critique has been selected up to now. Furthermore, we assume that $ri = ei_8$. Then the resulting set of new candidate items $CI = \{ci_3, ci_4, ci_7\}$. In this case, item ci_4 has by far the highest quality value and thus would be the item r returned by ItemRecommend – $quality(ci_4, ei_8, cc_2{:}>_{zoom(9x)}, <_{price(278)} = 0.61)$. This item selection approach helps take into account already selected critiques – that is, preferences already specified are not ignored in future critiquing cycles. Further

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Table 4.14. *Dynamic critiquing:* quality *of candidate items* $CI = \{ci_3, ci_4, ci_7\}$ *with regard to* $ri = ei_8$ *and* $CC_U = \{cc_2 :>_{zoom(9x)}, <_{price(278)}\}$.

candidate item ci	compatibility(ci, CC_U)	similarity(ci, ri)	*quality*
ci_3	1.0	0.40	0.40
ci_4	1.0	0.61	0.61
ci_7	1.0	0.41	0.41

related item recommendation approaches are discussed by Reilly et al. (2004), who focus especially on the issue of consistency in histories of already selected critiques. For example, if the user has initially specified the upper bound for the price with $<_{price(150)}$ and later specifies $>_{price(300)}$, one of those critiques must be removed from the critique history to still have available candidate items for the next critiquing cycle.

4.4.5 Critique diversity

Compound critiques are a powerful mechanism for effective item search in large assortments – especially for users who are nonexperts in the corresponding product domain. All the aforementioned critiquing approaches perform well as long as there are no "hot spots," in which many similar items are concentrated in one area of the item space. In such a situation a navigation to other areas of the item space can be very slow. Figure 4.6 depicts such a situation, in which a compound critique on *price* and *mpix* leads to recommended items that are quite similar to the current one.

An approach to avoid such situations is presented by McCarthy et al. (2005), who introduce a quality function (see Formula 4.9) that prefers compound critiques (cc) with low support values (many items can be eliminated) that are at the same time diversified from critiques CC_{Curr} already selected for presentation in the current critiquing cycle.

$$quality(cc, CC_{Curr}) = support(cc) * overlap(cc, CC_{Curr}) \qquad (4.9)$$

The support for a compound critique cc corresponds to the support of the corresponding association rule – for example, support(ar_3) = support(cc_3) = 85.7 (see Table 4.13). The overlap between the currently investigated compound critique cc and CC_{Curr} can be calculated on the basis of Formula 4.10. This formula determines the overlap between items supported by cc and items supported by critiques of CC_{Curr} – that is, $items(CC_{Curr})$ denotes the items

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

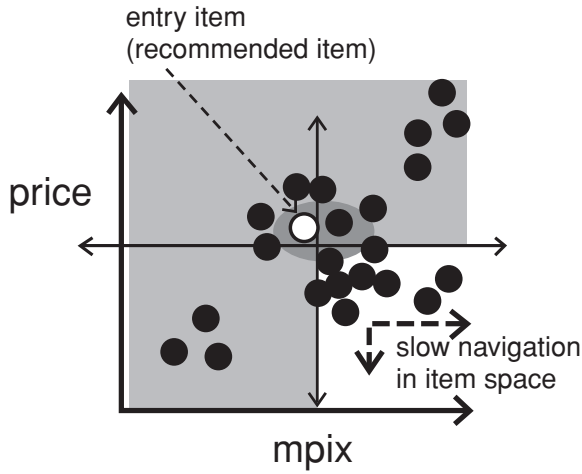

Figure 4.6. Critique-based navigation: slow navigation in dense item spaces.

accepted by the critiques in CC_{Curr}.

$$overlap(c, CP) = \frac{|items(\{cc\}) \cap items(CC_{Curr})|}{|items(\{cc\}) \cup items(CC_{Curr})|} \tag{4.10}$$

The lower the value of the function *quality* (preferred are low support and low overlap to already presented critiques), the higher the probability for a certain critique to be presented in the next critiquing cycle. Table 4.15 depicts the result of applying Formula 4.9 to $CC_{Curr} = \{\text{cc}_2\text{: } >_{zoom(9\times)},\ <_{price(278)}\}$ and two candidate association rules $\{ar_2, ar_3\}$ (see Table 4.13). Conforming to this formula, the quality of compound critique cc_1 is higher than the quality of cc_3.

Table 4.15. *Dynamic critiquing: quality of compound critiques* $CC = \{cc_1, cc_3\}$ *derived from association rules* $AR = \{ar_1, ar_3\}$ *assuming that* $CC_{Curr} = \{cc_2\text{: } >_{zoom(9\times)},\ <_{price(278)}\}$.

compound critiques (CC)	support(cc)	overlap(cc, CP)	*quality*
cc_1: $>_{mpix(9.1)}$, $>_{zoom(9\times)}$	28.6	66.7	0.19
cc_3: $=_{movies(yes)}$, $<_{price(278)}$	85.7	50.0	0.43

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

4.5 Example applications

In the final part of this chapter we take a more detailed look at two commercial recommender applications: a constraint-based recommender application developed for a Hungarian financial service provider and a case-based recommendation environment developed for recommending restaurants located in Chicago.

4.5.1 The VITA constraint-based recommender

We now move from our working example (digital camera recommender) to the domain of financial services. Concretely, we take a detailed look at the VITA financial services recommender application, which was built for the Fundamenta loan association in Hungary (Felfernig et al. 2007b). VITA supports sales representatives in sales dialogs with customers. It has been developed on the basis of the CWAdvisor recommender environment presented by Felfernig et al. (2006)

Scenario. Sales representatives in the financial services domain are challenged by the increased complexity of service solutions. In many cases, representatives do not know which services should be recommended in which contexts, and how those services should be explained. In this context, the major goal of financial service providers is to improve the overall productivity of sales representatives (e.g., in terms of the number of sales dialogs within a certain time period or number of products sold within a certain time period) and to increase the advisory quality in sales dialogs. Achieving these goals is strongly correlated to both an increase in the overall productivity and a customer's interest in long-term business connections with the financial service provider.

Software developers must deal with highly complex and frequently changing recommendation knowledge bases. Knowledge-based recommender technologies can improve this situation because they allow effective knowledge base development and maintenance processes.

The Fundamenta loan association in Hungary decided to establish knowledge-based recommender technologies to improve the performance of sales representatives and to reduce the overall costs of developing and maintaining related software components. In line with this decision, Fundamenta defined the following major goals:

- *Improved sales performance*: within the same time period, sales representatives should be able to increase the number of products sold.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

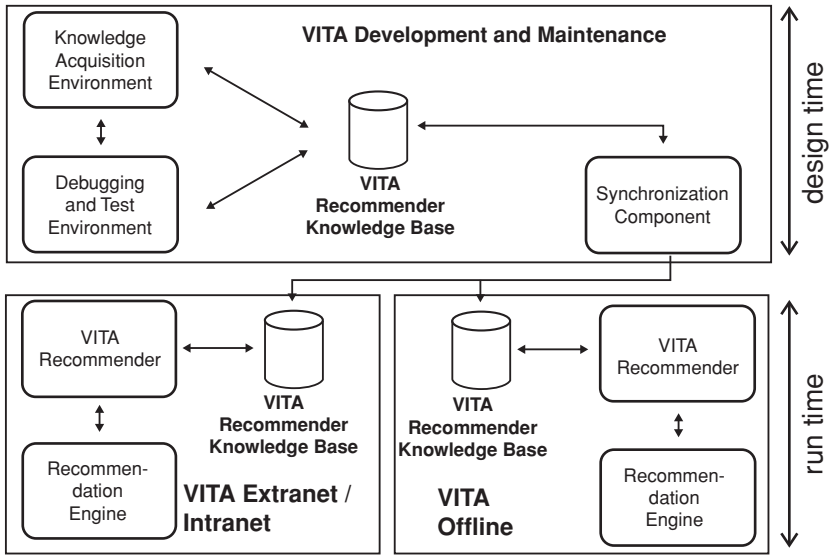

Figure 4.7. Architecture of the VITA sales support environment (Felfernig et al. 2007).

- *Effective software development and maintenance*: the new technologies should ease the development of sales knowledge bases.

Application description. The resulting VITA sales support environment (Figure 4.7) supports two basic (and similar) advisory scenarios. On one hand, VITA is a web server application used by Fundamenta sales representatives and external sales agents for the preparation and conducting of sales dialogs. On the other hand, the same functionality is provided for sales representatives using their own laptops. In this case, new versions of sales dialogs and knowledge bases are automatically installed when the sales representative is connected with the Fundamenta intranet.

For both scenarios, a knowledge acquisition environment supports the automated testing and debugging of knowledge bases. Such a knowledge base consists of the following elements (see Section 4.3):

- *Customer properties*: each customer must articulate his or her requirements, which are the elementary precondition for reasonable recommendations. Examples for customer properties in the financial services domain are *age*, *intended run time of the service*, *existing loans in the portfolio*, and the like.

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

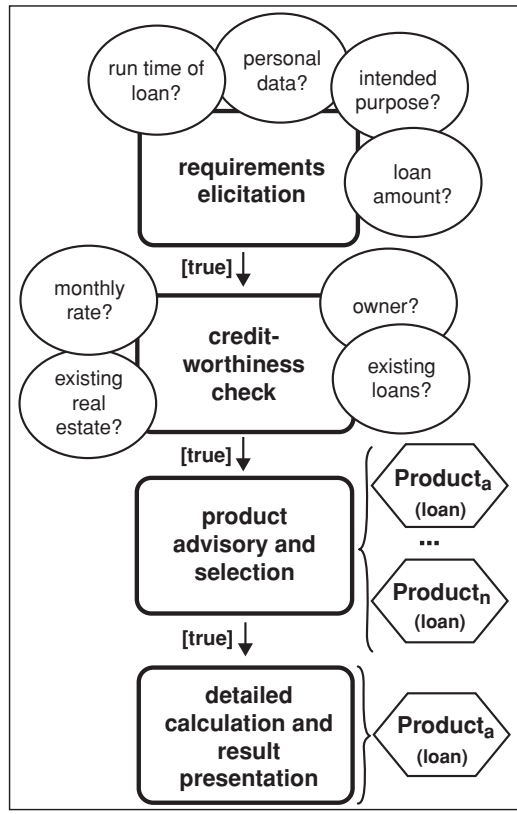

Figure 4.8. Example of advisory process definition (loan advisory) (Felfernig et al. 2007b).

- *Product properties and instances*: each product is described in terms of a set of predefined properties such as *recommended run time*, *predicted performance*, *expected risk*, and so on.
- *Constraints*: restrictions that define which products should be recommended in which context. A simple example of such a constraint is *customers with a low preparedness to take risks should receive recommendations that do not include high-risk products*.
- *Advisory process definition*: explicit definitions of sales dialogs are represented in the form of state charts (Felfernig and Shchekotykhin 2006) that basically define the context in which questions should be posed to the user (an example of a simple advisory process definition is depicted in Figure 4.8).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

In Fundamenta applications, recommendation calculations are based on the execution of conjunctive queries (for an example, see Section 4.3). Conjunctive queries are generated directly from requirements elicited within the scope of a recommendation session. The recommendation process follows the advisory process definition.

A loan recommendation process is structured in different phases (*requirements elicitation*, *creditworthiness check*, *product advisory/selection*, and *detailed calculation/result presentation*). In the first phase, basic information regarding the customer (*personal data*) and the major purpose of and requirements regarding the loan (e.g., *loan amount*, *run time of loan*) are elicited. The next task in the recommendation process is to check the customer's creditworthiness on the basis of detailed information regarding the customer's current financial situation and available financial securities. At this time, the application checks whether a solution can be found for the current requirements. If no such solution is available (e.g., too high an amount of requested money for the available financial securities), the application tries to determine alternatives that restore the consistency between the requirements and the available set of products. After the successful completion of the phase *creditworthiness check*, the recommender application proposes different available loan alternatives (redemption alternatives are also taken into account). After selecting one of those alternatives, the recommendation process continues with a detailed calculation of specific product properties, such as the monthly redemption rates of the currently selected alternative. A screen shot of the VITA environment is depicted in Figure 4.9.

Knowledge acquisition. In many commercial recommender projects, generalists who possess deep domain knowledge as well as technical knowledge about recommender technologies are lacking. On one hand, knowledge engineers know how to create recommender applications; on the other hand, domain experts know the details of the product domain but do not have detailed technical knowledge of recommenders. This results in a situation in which technical experts have the responsibility for application development and domain experts are solely responsible for providing the relevant product, marketing, and sales knowledge. This type of process is error-prone and creates unsatisfactory results for all project members.

Consequently, the overall goal is to further improve knowledge-based recommender technologies by providing tools that allow a shift of knowledge base development competencies from knowledge engineers to domain experts. The knowledge acquisition environment (CWAdvisor [Felfernig et al. (2006)]) that is used in the context of VITA supports the development of recommender

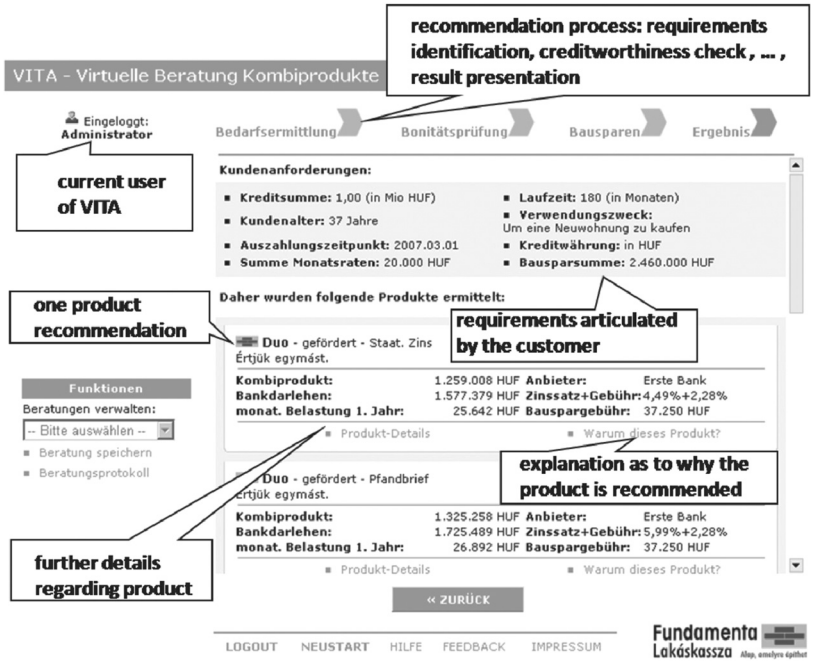

Figure 4.9. Screen shot of VITA sales support environment (Felfernig et al. 2007b).

knowledge bases and recommender process definitions on a graphic level. A glimpse of the basic functionalities of this acquisition is given in Figure 4.10. Different recommender applications can be maintained in parallel – for example, investment and financing recommenders in the financial services domain. Each of those recommenders is defined by a number of product properties, customer properties, and constraints that are responsible for detecting inconsistent customer requirements and for calculating recommendations.

A simple example of the definition of constraints in the financial services domain is given in Figure 4.11. This constraint indicates that high rates of return require a willingness to take risks. The CWAdvisor environment (Felfernig et al. 2006) supports rapid prototyping processes by automatically translating recommender knowledge bases and process definitions into a corresponding executable application. Thus customers and engineers are able to immediately detect the consequences of changes introduced into the knowledge base and the corresponding process definition.

Ninety percent of the changes in the VITA knowledge base are associated with the underlying product assortment because of new products and changing

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

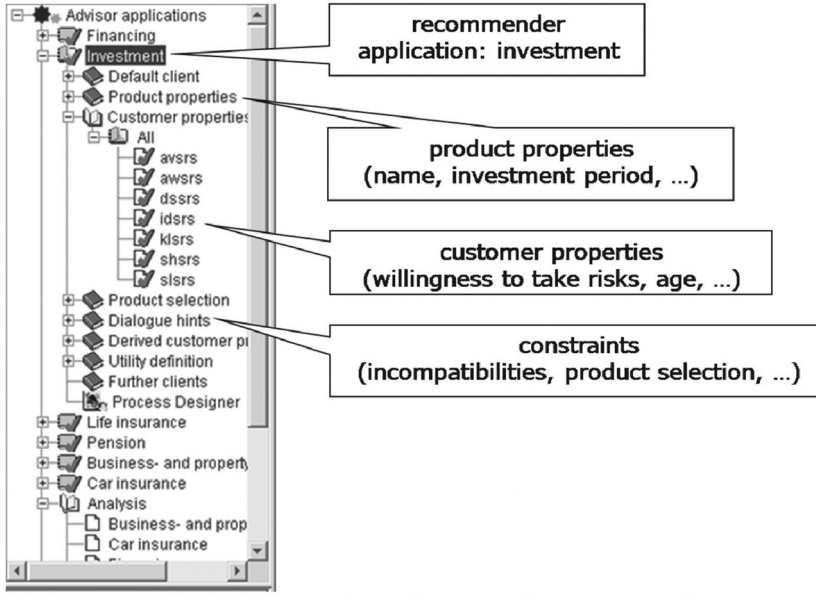

Figure 4.10. Knowledge acquisition environment (Felfernig et al. 2006).

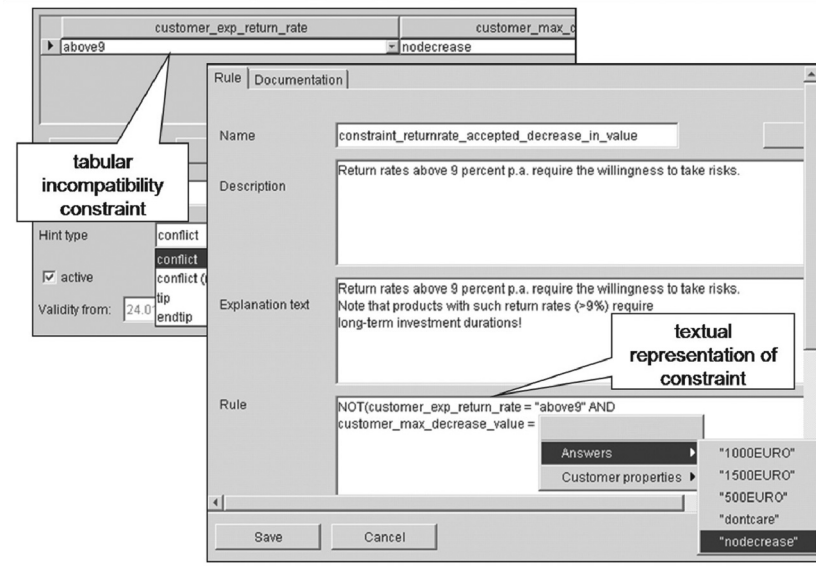

Figure 4.11. Example of incompatibility constraint: high return rates are incompatible with low preparedness to take risks (Felfernig et al. 2006).

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

interest rates (Felfernig et al. 2007b). Change requests are collected centrally and integrated into the VITA recommender knowledge base once a month. The remaining 10 percent of the changes are related to the graphical user interface and to the explanation and visualization of products. These changes are taken into account in new versions of the recommender application, which are published quarterly.

4.5.2 The Entree case-based recommender

A well-known example of a critiquing-based commercial recommender application is Entree, a system developed for the recommendation of restaurants in Chicago (Burke 2000, Burke et al. 1997). The initial goal was to guide participants in the 1996 Democratic National Convention in Chicago, but its success prolonged its usage for several years.

Scenario. Restaurant recommendation is a domain with a potentially large set of items that are described by a predefined set of properties. The domain is complex because users are often unable to fully define their requirements. This provides a clear justification for the application of recommendation technologies (Burke et al. 1997). Users interact with Entree via a web-based interface with the goal of identifying a restaurant that fits their wishes and needs; as opposed to the financial services scenario discussed in Section 4.5.1, no experts are available in this context who support users in the item retrieval process. FindMe technologies introduced by Burke et al. (1997) were the major technological basis for the Entree recommender. These technologies implement the idea of critique-based recommendation that allows an intuitive navigation in complex item spaces, especially for users who are not experts in the application domain.

Application description. A screenshot of an Entree-type system is shown in Figure 4.12.

There are two entry points to the system: on one hand, the recommender can use a specific reference restaurant as a starting point (preselected on the basis of textual input and a text-based retrieval (Burke et al. 1997)); on the other hand, the user is able to specify the requirements in terms of typical restaurant properties such as *price*, *cuisine type*, or *noise level* or in terms of high-level properties such as *restaurant with a nice atmosphere* (Burke et al. 1997). High-level properties (e.g., *nice atmosphere*) are translated to low-level item properties – for example, *restaurant with wine cellar and quiet location*. High-level properties

Figure 4.12. Example critiquing-based restaurant recommender.

can be interpreted as a specific type of compound critique, as they refer to a collection of basic properties. The Entree system strictly follows a static critiquing approach, in which a predefined set of critiques is available in each critique cycle. In each cycle, Entree retrieves a set of candidate items from the item database that fulfill the criteria defined by the user (Burke 2002a). Those items are then sorted according to their similarity to the currently recommended item, and the most similar items are returned. Entree does not maintain profiles of users; consequently, a recommendation is determined solely on the basis of the

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

Figure 4.13. Example of critiquing-based restaurant recommender: result display after one critiquing cycle.

currently displayed item and the critique specified by the user. A simple scenario of interacting with Entree-type recommender applications follows (see Figure 4.13).

The user starts the interaction with searching for a known restaurant, for example, the Biergasthof in Vienna. As shown in Figure 4.12, the recommender manages to identify a similar restaurant named *Brauhof* that is located in the city of Graz. The user, in principle, likes the recommended restaurant but

https://doi.org/10.1017/CBO9780511763113.006 Published online by Cambridge University Press

would prefer a less expensive one and triggers the *Less $$* critique. The result of this query is the *Brau Stüberl* restaurant in city of Graz, which has similar characteristics to *Brauhof* but is less expensive and is now acceptable for the user.

Knowledge acquisition. The quality of a recommender application depends on the quality of the underlying knowledge base. When implementing a case-based recommender application, different types of knowledge must be taken into account. A detailed description of the cases in terms of a high number of item attributes requires investing more time into the development of the underlying similarity measures (Burke 2002a). In Entree, for each item a corresponding local similarity measure is defined that explains item similarity in the context of one specific attribute. For example, two restaurants may be very similar in the dimension *cuisine* (e.g., both are Italian restaurants) but may be completely different in the dimension *price*. The global similarity metric is then the result of combining the different local similarity metrics. An important aspect in this context is that similarity metrics must reflect a user's understanding of the item space, because otherwise the application will not be successful (Burke 2002a). Another important aspect to be taken into account is the quality of the underlying item database. It must be correct, complete, and up to date to be able to generate recommendations of high quality. In the restaurant domain, item knowledge changes frequently, and the information in many cases has to be kept up to date by humans, which can be costly and error-prone.

4.6 Bibliographical notes

Applications of knowledge-based recommendation technologies have been developed by a number of groups. For example, Ricci and Nguyen (2007) demonstrate the application of critique-based recommender technologies in mobile environments, and Felfernig and Burke (2008) and Felfernig et al. (2006–07) present successfully deployed applications in the domains of financial services and consumer electronics. Burke (2000) and Burke et al. (1997) provide a detailed overview of knowledge-based recommendation approaches in application domains such as restaurants, cars, movies, and consumer electronics. Further well-known scientific contributions to the field of critiquing-based recommender applications can be found in Lorenzi and Ricci (2005), McGinty and Smyth (2003), Salamo et al. (2005), Reilly et al. (2007a), and Pu et al. (2008). Felfernig and Burke (2008) introduce a categorization of principal recommendation approaches and provide a detailed overview of constraint-based

recommendation technologies and their applications. Zanker et al. (2010) formalize different variants of constraint-based recommendation problems and empirically compare the performance of the solving mechanisms. Jiang et al. (2005) introduce an approach to multimedia-enhanced recommendation of digital cameras, in which changes in customer requirements not only result in a changed set of recommendations, but those changes are also animated. For example, a change in the personal goal from portrait pictures to sports photography would result in a lens exchange from a standard lens to a fast lens designed especially for the high-speed movements typical in sports scenes. Thompson et al. (2004) present a knowledge-based recommender based on a combination of knowledge-based approaches with a natural language interface that helps reduce the overall interaction effort.