© 2021 World Scientific Publishing Company
https://doi.org/10.1142/9789811239922_0002

Chapter 2

AI Techniques for Software Requirements Prioritization

Alexander Felfernig
Institute of Software Technology
Graz University of Technology, Austria

2.1 Introduction

Limited resources, market demands, and technical restrictions regarding the implementation of software features often demand for the prioritization of requirements [1–4]. The focus of prioritization is the *ranking and selection of requirements that should be included in future software releases.* Intelligent decision support in prioritization is extremely important since especially when dealing with large assortments of requirements, manual prioritization processes tend to become very costly [5–8]. Potential sub-optimal prioritizations can lead to different negative effects such as *waste of time* due to a focus on irrelevant requirements, *opportunity costs* due to the fact that the relevant features are not provided first, and *missing focus on market demands* that could lead in the worst case to total loss [9]. In this context, prioritization can take place on the strategic level as well as an on the operative level, which is typically associated with short-term prioritization tasks [10, 11]. The prioritization approaches discussed in this chapter are based on AI techniques from the areas of constraint reasoning & optimization [12], utility-based recommendation [13], content-based recommendation [14], matrix factorization [15], conflict detection [16], and model-based diagnosis [17].

An overview of different prioritization tasks is given in Fig. 2.1. This categorization is based on two dimensions. First, *level of requirements*

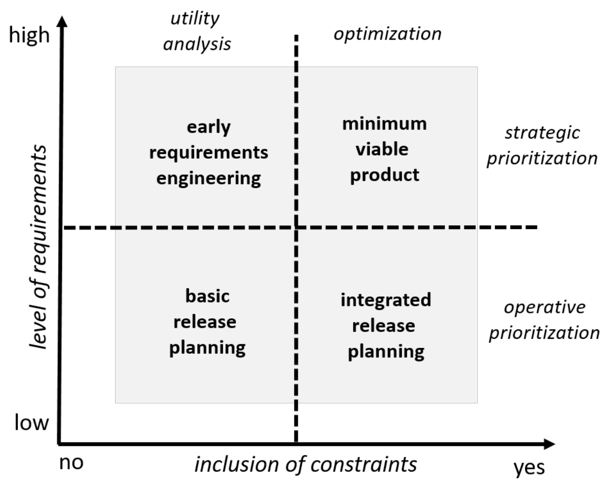

Fig. 2.1 Prioritization variants in software development contexts.

specifies the granularity of requirements specifications, i.e., to which extent these requirements can already be translated into corresponding detailed software features. Second, *inclusion of constraints* refers to which extent relationships between requirements and relationships to external factors are taken into account in the prioritization process. Examples of constraints (dependencies) between requirements are *x requires y* (*x* must not be implemented before *y*) and *x excludes y* (only one of these requirements should be implemented). Examples of external factors are the available budget for a software project, available personnel resources, and specific preferences of stakeholders engaged in a software project. Along with these two dimensions, there exist different prioritization approaches, which can be differentiated with regard to the *granularity level of requirements* and the *degree of the inclusion of constraints.*

Early requirements engineering is related to the idea of figuring out the requirements that have the highest importance, for example, for the market or specific customer communities. Prioritization tasks typically refer to high-level requirements, furthermore, no specific constraints are included. The major focus is to figure out the most relevant features of a product with a market relevance. Requirements in such scenarios can be regarded as high-level, for example, *"the new e-learning software should include a motivation functionality that persuades students to intensively learn the course topics"* or *"the new e-learning software should support natural language based interaction mechanisms"*.

A *minimal viable product* (MVP) should include a minimal set of features that can be integrated as parts of a fully operable software offered to customers. MVPs are a typical approach to get to the market as soon as possible with the most relevant features of a software. In this context, constraints play an important role since the prioritization of requirements has to take into account constraints such as available personnel and budget resources. Requirements can also be regarded as high-level and constraints primarily refer to available budgets and personnel resources. Examples of constraints are *"motivation features of the new e-learning software should solely include the aspect of social influence"* and *"for the first version of the software, natural language interaction should support the answering of multiple-choice questions with single correct answers"*.

Basic release planning does not fully take into account further constraints such as available budget, personnel resources, and time restrictions regarding the implementation of requirements. This type of prioritization covers implementation scenarios where releases are planned on an operational level without taking into account in detail constraints regarding available personnel and budget resources as well as time limitations. Examples of requirements in such contexts are *"the basic scenario for a social influence based persuasion is the following ... the user interface implementation of this function should look like as follows ..."* or *"the basic scenario for supporting multiple choice questions in the context of natural language interactions is the following ... the user interface implementation can be sketched as follows ..."*. On a technical level, basic release planning can be performed using approaches similar to those used in the context of early requirements engineering.

Finally, *integrated release planning* represents a full-fledged release planning [18,19] on the basis of detailed constraints representing organizational data and rules. In this context, both, constraints regarding dependencies between requirements as well as constraints related to external factors are taken into account. Similar to basic release planning, requirements are defined with fine granularity. A major difference between basic release planning and integrated release planning is the availability of more detailed constraint information, for example, integrated release planning is able to take into account the individual availability of developers (in terms of engagement in other projects and presence or absence during specific time periods). Furthermore, dependencies between requirements can be taken into account on a formal level.

On the level of prioritization techniques, there are *two basic approaches* to support prioritization processes — see Achimugu *et al.* [1]. *First*, prioritization can be regarded as an *optimization task* where the objective is to identify a prioritization that takes into account the preferences of individual stakeholders and also helps to optimize the prioritization with regard to a set of predefined constraints [20]. On a technical level, optimization-based prioritization is often based on a hybrid approach where the identification and aggregation of stakeholder preferences is supported by utility analysis [21–23] and optimization is performed on the basis of constraint reasoning [12, 24].

Utility-based approaches focus on an analysis of the given requirements with regard to a set of *interest dimensions* and less on automated optimization. Different variants of this approach can be implemented, for example, a utility-based ranking can be extended with the concepts of liquid democracy [25]. Finally, social networks can be exploited as data sources for the identification of new requirements, which are regarded as relevant by the underlying social network [26]. In terms of the application of the mentioned prioritization techniques, early requirements engineering and basic release planning focus more on utility-based prioritization approaches whereas minimum viable product and integrated release planning focus on optimization-based prioritization approaches.

The major contributions of this chapter are the following. First, we provide an overview of existing techniques that help to improve the quality of prioritization processes in requirements engineering. Second, we show the application of these techniques in the context of working examples. Third, in order to stimulate further work in related fields, we discuss relevant issues for future work.

The remainder of this chapter is organized as follows. Sections 2.2–2.5 include a discussion of the application of AI techniques in the scenarios of *early requirements engineering*, *minimum viable products*, *basic release planning*, and *integrated release planning.* These sections include a description of the underlying scenarios and working examples. Section 2.6 provides insights how to support stakeholder selection, which is an important issue when it comes to the assignment of requirement validation tasks. Section 2.7 provides an overview of issues for future research. Finally, we conclude the chapter with Section 2.8.

2.2 Early Requirements Engineering

A basic means to support prioritization tasks in early requirements engineering is to perform a utility analysis of a given set of requirements. *Utility-based prioritization* is based on the concepts of *multi-attribute utility theory* [27] — different variants thereof are possible. First, individual requirements are evaluated with regard to *interest dimensions* (e.g., risk level of a requirement and the commercial relevance of a requirement). The utility of the requirement is then determined on the basis of the sum of interest dimension specific utility values. Interest dimensions can be associated with a weight, for example, *low risks are more important than high profits*. Utility-based prioritization can also be implemented on the basis of analytic hierarchy process (AHP) [28]. A major disadvantage of this approach is that requirements have to be evaluated pairwise which does not scale well when the number of requirements increases.

Interest dimensions, i.e., basic evaluation criteria for utility-based prioritization can differ depending on the underlying decision scenario. Examples of such interest dimensions in company-related software projects are *effort to implement a requirement*, *risk of not being able to implement a requirement*, and *business relevance of a requirement (profit)* [1]. In open source settings, the dimensions can be different since open source contributors have to decide individually on which requirement to work next. Examples of related interest dimensions could be *personal expertise of an open source developer* and *importance of a requirement for the community* [29].

Utility analysis supports stakeholders in the prioritization of requirements with regard to a set of interest dimensions $D = \{d_1, d_2, ..., d_n\}$. The underlying idea is that requirements are first analyzed by individual stakeholders (also denoted as users) — see Tables 2.1–2.2. Such decisions are often group decisions where stakeholders are in charge of prioritizing a set of requirements [30].

In this simplified example, *users* are in charge of evaluating the requirements $req_1..req_5$ with regard to the interest dimensions *business relevance* and *risk*. Thereafter, individual evaluations are aggregated to determine the utility of requirements. In this context, Formula 2.1 can be used to calculate the utility of a requirement with regard to a specific interest dimension d. Furthermore, Formula 2.2 is used to determine the overall utility of a requirement.

Table 2.1 Evaluation of the dimension *relevance* (high rating = high relevance).

	user_1	user_2	user_3	user_4
req_1	1	4	5	2
req_2	10	6	1	7
req_3	2	6	5	2
req_4	1	1	3	7
req_5	7	8	6	5

Table 2.2 Evaluation of the dimension *risk* (high rating = low risk).

	user_1	user_2	user_3	user_4
req_1	2	7	3	2
req_2	9	9	1	7
req_3	2	10	3	2
req_4	2	5	3	1
req_5	3	2	3	5

$$utilityreq(req, d) = \frac{\Sigma_{u \in Users} eval(req, d, u)}{|Users|} \tag{2.1}$$

$$utility(req) = \frac{\Sigma_{d \in Dims} utilityreq(req, d) \times weight(d)}{|Dims|} \tag{2.2}$$

The determined utilities are then encoded in a ranking (see Table 2.3).

Table 2.3 Prioritization of requirements $req_1..req_5$ with regard to the interest dimensions *relevance* (weight = 0.75) and *risk* (weight = 0.25).

requirement req_i	req_1	req_2	req_3	req_4	req_5
utility(req_i)	4.63	**5.75**	4.06	2.94	4.56
priority(req_i)	2	1	4	5	3

The presented approach to group-based multi attribute utility analysis [30] is based on the assumption that each stakeholder is able to provide feedback on each of the given requirements. This might not be possible for various reasons, for example, stakeholders are simply not available, i.e., do not have time or they might have issues in terms of missing knowledge needed to evaluate a requirement. In such cases, mechanisms are needed to be able to transfer votes in a flexible fashion. Such an approach to liquid-democracy based prioritization is introduced in [25]. The major difference compared to the aforementioned approach is that individual stakeholders are allowed to vote more than once and to transfer their votes to other stakeholders.

An alternative approach to handle *missing values* in requirements evaluation is to apply machine learning concepts, which help to automatically complete a potentially sparse rating matrix [15]. The automatically determined requirements evaluations can then be proposed to stakeholders and can also serve as indicators of potential issues related to contradictory evaluations, which have to be resolved. Table 2.4 depicts a user-item matrix, which includes a couple of missing evaluations (denoted with "?").

Table 2.4 Association of users with requirements $req_1..req_5$.

relevance	user$_1$	user$_2$	user$_3$	user$_4$
req$_1$	?	?	5	?
req$_2$	10	?	1	?
req$_3$	?	6	?	2
req$_4$	?	?	3	?
req$_5$	?	?	?	5

Based on the information included in Table 2.4, we can perform so-called dimensionality reduction and describe the relationship between users and requirements in terms of two low-dimensional matrices U and R where the former describes the relationship between users and abstract dimensions (hidden features) (see Table 2.5) and the latter the relationship between items and abstract dimensions (see Table 2.6).

Table 2.5 User × interest dimension ($d_1..d_3$) affinity matrix U.

	user$_1$	user$_2$	user$_3$	user$_4$
d$_1$	3,652807135	1,251029912	0,148850849	1,870385191
d$_2$	2,406538532	1,830201936	1,766613942	0
d$_3$	0,053547355	0,176813763	1,86544824	0,002507298

Table 2.6 Requirement × interest dimension ($d_1..d_3$) affinity matrix R.

	d$_1$	d$_2$	d$_3$
req$_1$	0,318390415	0,359262854	2,305033956
req$_2$	2,527786478	0,3177104899	0,035500999
req$_3$	1,072394897	2,524779729	0,126403403
req$_4$	0,167185814	1,181561695	0,467019398
req$_5$	2,665424355	0,109392275	0,008631143

The table entries can be learned on the basis of a matrix factorization approach that is based on non-linear optimization. The optimization goal is to find values for the low-dimensional tables, which help to predict the missing table entries as good as possible. For a detailed discussion of matrix factorization techniques we refer to [15].

Similar to the description of the relationship between users and hidden features, we can describe the relationship between requirements and hidden features. The higher the value, the higher the corresponding affinity between users (requirements) and the corresponding hidden features. We want to emphasize that in the matrix factorization context features are *hidden*, i.e., it is not clear if and which hidden feature corresponds to a specific evaluation dimension (as discussed in the context of utility-based prioritization).

The two low-dimensional matrices U and R can now be used to calculate a prediction for an unspecified user $\times$ requirement pair denoted with "?" (see Table 2.4). By applying matrix multiplication, we can, for example, determine a prediction of the evaluation of requirement req_1 by $user_1$. The corresponding table entry results from the expression $0,318390415 \times 3,652807135 + 0,359262854 \times 2,406538532 + 2,305033956 \times 0,053547355$ which is $2,151027154$. Expecting predictions on a scale 0..10, the prediction for the evaluation of requirement req_1 by $user_1$ appears to be rather low.

2.3 Minimum Viable Products

Minimum viable products (MVPs) represent products (in our case software components) that include a minimum set of requirements applicable and of value for a customer. In the context of software development, MVP development is extremely important especially for start-up companies since resources are often extremely limited and there is only one chance to develop the right product for the customer community. Consequently, prioritization support is extremely important in such scenarios. MVP development is related to DevOps software processes which are characterized by extensive automation and continuous updates [31]. Such processes support a more in-depth customer integration into feedback and prioritization and — as a consequence — help to increase the quality of prioritization due to deeper insights into the progress of the project.

Prioritizations for *minimum viable products* typically have to deal with high-level requirements, which do not describe specific functionalities but rather generic features of the software. For these features, it should be made clear which are the most relevant ones that can realistically be implemented. We can consider the task of selecting a subset of requirements to be included in a minimal viable product as a utility-based prioritization task where requirement *utilities* and *time estimates* are used as basic inputs in a follow-up process that focuses on optimizing the selection of a bundle of most relevant features (requirements). Thus, MVP-oriented prioritization supports a kind of triage process [32] where the most important and feasible requirements are implemented first.

Formula 2.3 restricts the available time resources, i.e., how much time is available to implement the new MVP features. In typical start-up scenarios, this would reflect a situation where, for example, four persons together can spend around one month to implement market-relevant features into an MVP. To make good use of the available time, resource planning can be used to calculate an optimal subset of requirements to be included ($included(req_i)$) in the MVP. An example of how to take into account time restrictions is shown in Formula 2.3.

$$time(req_1) \times included(req_1) + .. + time(req_n) \times included(rec_n) \leq maxtime \tag{2.3}$$

The overall optimization objective of this resource planning task is expressed with Formula 2.4. The utility of the selected requirements (requirements, which should be part of the MVP) should be maximized while taking into account additional restrictions (see Formula 2.3).

$$max \leftarrow utility(req_1) \times included(req_1) + .. + utility(req_n) \times included(rec_n) \tag{2.4}$$

Table 2.7 Selecting the most relevant requirements under given time conditions resulting in a maximum utility of 10.31 = utility(req_2)+utility(req_5).

requirement req_i	req$_1$	req$_2$	req$_3$	req$_4$	req$_5$
utility(req_i)	4.63	**5.75**	4.06	2.94	**4.56**
time(req_i)	3	4	4	3	5
selected	0	**1**	0	0	**1**

2.4 Basic Release Planning

Basic release planning follows a prioritization approach where requirements formulated on a fine-granular level are selected with regard to their relevance of being part of one of the next n releases — in the case of $n = 1$, this scenario is also denoted as *next release problem.* In most of the cases, such scenarios do not need the support of a high-sophisticated release planning solution. Example reasons for choosing a lightweight process are the *unavailability of resource data* required by release planning tools (e.g., data about resources already occupied in projects) and *limited budgets and personnel resources* to purchase and support a heavy-weight release planning software and to integrate this software with resource-related data sources.

Basic release planning focuses on the prioritization of requirements formulated on a fine-granular level. Initially, this process is often performed on the basis of a utility analysis (see Section 2.2). On the basis of the results of a utility analysis, stakeholders can propose assignments of requirements to releases. If a company's software process follows a *next release* strategy, i.e., the planning horizon is the next release, the corresponding selection task is to figure out the most relevant requirements for the next release. Basic release planning typically does not take into account constraints regarding available resources — such constraints are taken into account informally.

Tools supporting basic release planning can help to repair inconsistencies in the stakeholders' preferences regarding the assignment of requirements to releases. A scenario in the context of basic release planning is the following (see Table 2.8). Stakeholders (users) define their individual preferences regarding the assignment of requirements to releases. Since stakeholders can do this remotely and are initially often not allowed to see the preferences of other stakeholders, conflicts regarding defined release assignment preferences can occur [33].

Table 2.8 Preferences of stakeholders with regard to release assignments.

	$user_1$	$user_2$	$user_3$	$user_4$
req_1	1	1	2	1
req_2	2	2	3	3
req_3	3	3	3	3
req_4	1	2	2	3
req_5	4	1	1	1

In this context, constraint-based optimization can be applied to minimize the need of preference change per user (see Formula 2.7). We assume the existence of variables $ureq_{ij}$ with the domain 1..4 representing the releases 1..4, for example, $ureq_{11} = 1$ indicates that $user_1$ prefers the assignment of req_1 to release 1. Furthermore, we assume the existence of variables $ureq'_{ij}$, which represent the solution space. The constraint

$ureqcount_{ij} = abs(ureq_{ij} - ureq'_{ij})$ indicates whether a user preference has to be adapted. Furthermore, we need to count the number of changes needed per user i (see Formula 2.5). The number of preference changes per user i is represented by variable chn_i (see Formula 2.5).

$$chn_i \leftarrow ureqcount_{i1} + .. + ureqcount_{in} \tag{2.5}$$

Furthermore, we want to assure *consensus*, i.e., each requirement j has to be assigned to exactly one release (see Formula 2.6).

$$ureq'_{1j} = .. = ureq'_{mj} \tag{2.6}$$

Given this knowledge, we can define an optimization problem with the overall goal to minimize the number of changed release assignments while at the same time being fair, i.e., it should not be the case that (in the worst case) all needed changes are affecting a single stakeholder. This criteria is represented by Formula 2.7. The underlying idea is that the pairwise distance between stakeholders in terms of the number of needed stakeholder-specific preference adaptations should be minimized.

$$min \leftarrow abs(chn_1 - chn_2) + .. + abs(chn_{n-1} - chn_n) \tag{2.7}$$

Formula 2.8 represents an alternative optimization function where the expected solution represents a tradeoff between *fairness* among stakeholders in terms of a fair share of individual changes of preferences and *minimality* in terms of the overall number of needed changes.

$$min \leftarrow (abs(chn_1 - chn_2) + .. + abs(chn_{n-1} - chn_n)) \times (chn_1 + .. + chn_n) \tag{2.8}$$

This kind of knowledge can be exploited by optimization features of constraint solvers such as CHOCO.[1]

2.5 Integrated Release Planning

On top of the concepts of basic release planning, integrated release planning has a strong focus on integrating additional constraints related to the dependency between requirements and constraints related to the availability of resources, limits of resource consumption, and the assignment of stakeholders to individual tasks. Integrated release planning requires detailed information about the assignment of employees to current projects and their availability. Furthermore, project-specific release plans have to be synchronized since employees can be assigned to multiple projects during

[1]choco-solver.org

the same time period. A special case are distributed project scenarios where a large project is conducted by different independent teams that work on some common features, which have to be taken into account in the release plans of the individual project partners.

Table 2.9 provides a representative overview of modeling concepts that can be used in the context of release planning. Requirements can be represented as basic components with associated properties represented as finite domain variables. For example, $req_1.rel$ denotes requirement req_1 with the associated release $req_1.rel$, which could be represented, for example, by the domain 1..3, i.e., the look-ahead factor for releases would be 3. Another example of a property which can be associated with a requirement req_i is $req_i.dur$, which denotes the time estimate for requirement req_i.

Table 2.9 Examples of basic constraints used for defining release planning tasks. In this context, req_i denotes a requirement, $req_i.rel$ denotes the corresponding release, and $req.dur$ denotes the estimated development time for a requirement.

Definition	Description
$req_i.rel = a$	req_i is assigned to release a
$req_i.rel < req_j.rel$	req_i must be implemented before req_j
$req_i.rel \leq req_j.rel$	req_j must not be implemented before req_i
$req_i.rel \neq req_j.rel$	req_i and req_j must have different releases
$req_i.rel \leq a$	implementation of req_i not after release a
$req_i.rel \geq a$	implementation of req_i not before release a
$req_i.rel = n \vee req_j.rel = n$	req_i or req_j not in release plan
$\neg(\lvert req_i.rel - req_j.rel\rvert > k)$	req_i and req_j must be implemented timely
$\lvert\{r \in R : r = rel\}\rvert \leq a$	not more than a requirements in release rel
$\Sigma_{r \in R \wedge r.rel=rel}(r.dur) \leq a$	not more than a hours bounded to rel

A simple example of the application of the modeling concepts shown in Table 2.9 is given in Tables 2.10–2.11. Table 2.10 includes dependencies between requirements that are considered correct and have to be taken into account, i.e., the constraints are so-called *hard constraints*. For example $req_1.rel < req_2.rel$ denotes the fact that the implementation of req_1 has to be completed before the implementation of req_2 can be started. Since these constraints are assumed to be taken into account, they have to be consistent, i.e., at least one solution should exist. Assuming a finite domain of 1..3 for each individual variable $req_i.rel$, a corresponding consistent variable assignment (solution) is $\{req_1.rel = 1, req_2.rel = 2, req_3.rel = 3, req_4.rel = 3, req_5.rel = 1\}$.

Please note that all constraint types shown in Table 2.8 can be either

Table 2.10 Example requirements and set D of corresponding dependencies. The domain of $req_i.rel$ is assumed to be 1..3.

	req$_1$.*rel*	req$_2$.*rel*	req$_3$.*rel*	req$_4$.*rel*	req$_5$.*rel*
req$_1$.*rel*	-	$<$	-	-	-
req$_2$.*rel*	-	-	$<$	-	$>$
req$_3$.*rel*	-	-	-	-	-
req$_4$.*rel*	-	-	-	-	$\neq$
req$_5$.*rel*	-	-	-	-	-

Table 2.11 Example set S of (inconsistent) stakeholder preferences.

	user$_1$	user$_2$	user$_3$	user$_4$
req$_1$.*rel*	$= 1$	$= 1$	≤ 2	$= 1$
req$_2$.*rel*	≥ 2	≥ 2	≥ 2	≥ 2
req$_3$.*rel*	≤ 2	≥ 2	$= 3$	≤ 3
req$_4$.*rel*	≥ 1	≥ 1	≥ 2	≥ 2
req$_5$.*rel*	≥ 2	$= 1$	$= 1$	≤ 2

represented as *hard constraints* or as *soft constraints* — in the context of our example, the entries of Table 2.10 are interpreted as hard constraints, those of Table 2.11 as soft constraints, i.e., stakeholder preferences that should be taken into account but could also be ignored in the case that not all stakeholder preferences could be taken into account. On the basis of the (hard) constraints shown in Table 2.10, stakeholders (users) can specify their individual preferences (see Table 2.11). For simplicity, we restrict the constraint type of user preferences to the form $req_i.rel = a$, $req_i.rel < a$, $req_i.rel > a$, $req_i.rel \leq a$, and $req_i.rel \geq a$.

The stakeholder preferences S in Table 2.11 are inconsistent. Detailed release planning can be regarded as an interactive process where stakeholders define their preferences and then try to establish consensus with regard to the final release plan. In the example shown in Table 2.11, the stakeholders have defined inconsistent preferences with regard to the requirements req_3 and req_5. More precisely, there is one set of conflicting preferences with regard to req_3 ($\{\{user_1 : (\leq 2), user_3 : (= 3)\}\}$) and two conflicting preferences with regard to req_5 ($\{\{user_1 : (\geq 2), user_2 : (= 1)\}, \{user_1 : (\geq 2), user_3 : (= 1)\}\}$). Combinations of preferences that induce an inconsistency are often denoted as conflict set [16, 17]. Conflict sets can be shown to stakeholders to indicate open issues and to stimulate discussions on how to resolve the existing inconsistencies. In our example, the inconsistent situation could be resolved if stakeholder $user_1$ would agree to change both of his (her) preferences. If we take into account both, the constraints in D and the preferences in S, we can detect two singleton conflicts both induced by the preferences of $user_1$ ($\{\{user_1 : (\leq 2)\}, \{user_1 : (\geq 2)\}\}$).

2.6 Stakeholder Recommendation

An issue in different prioritization scenarios is to figure out who should be in charge of validating a specific requirement since (s)he has the expertise needed. The quality of stakeholder/requirement assignment can have enormous impacts on the quality of a prioritization since sub-optimal evaluations can lead to sub-optimal prioritizations. Specifically, missing expertise can lead to situations where, for example, requirements of high relevance are evaluated as less relevant and — as a consequence — are not considered as a potential candidate for early releases. A major issue is to identify stakeholders who have the expertise and thus can provide reasonable evaluations of requirements. As sketched in Formula 2.9, expertise estimation can be implemented on the basis of the similarity between requirements already evaluated by a stakeholder and a set of new requirements.

Stakeholder expertise can be modeled in various ways. In the following, we provide a basic example of how to exploit the concepts of content-based recommendation [14] to propose reasonable assignments of stakeholders to requirements. Table 2.12 contains a set of new requirements with a corresponding set of keywords, which have been extracted from the requirement description. For these requirements, we would like to figure out automatically, which stakeholder would be the best one to work on this requirement, for example, to evaluate the requirement. Furthermore, Table 2.13 shows a list of stakeholders (users) and a corresponding list of keywords extracted from requirements descriptions the stakeholder worked on in the past. In order to estimate which stakeholder should work on which requirement, we can apply the concepts of content-based recommendation [14]. We can calculate the similarity between the keywords describing a stakeholder (see Table 2.13) and the keywords describing a requirement (see Table 2.12). This can be achieved by applying Formula 2.9 which helps to determine the stakeholder $\times$ requirements similarity.

$$sim(user, req) = \frac{2 \times |keywords(user) \cap keywords(req)|}{keywords(user) \cup keywords(req)} \tag{2.9}$$

The result of this similarity evaluation is summarized in Table 2.14. For req_1 and req_5, users with an average similarity have been identified as

Table 2.12 Requirements and keywords extracted from their descriptions.

Requirements	Keywords
req_1	registration users
req_2	basic payment
req_3	credit card payment
req_4	optimize user portfolio
req_5	optimize database

Table 2.13 Stakeholders and keywords of requirements they have validated.

Stakeholders	Keywords
$user_1$	registration feature database connection
$user_2$	payment process
$user_3$	credit card interfaces
$user_4$	credit card portfolio optimize

Table 2.14 Content-based similarity between stakeholders and requirements.

	$user_1$	$user_2$	$user_3$	$user_4$
req_1	**0.4**	0	0	0
req_2	0	**0.66**	0	0
req_3	0	0.5	**1.0**	**0.8**
req_4	0	0	0	**0.8**
req_5	**0.4**	0	0	**0.4**

candidates for validating the requirements. A user with a stronger similarity could be found for req_2. Finally, there is a strong similarity between requirements req_3, req_4, and $user_4$. Overall, $user_4$ seems to have a high coverage with regard to the potential requirements assignments. Finally, $user_3$ has the highest expertise with regard to a single requirement (req_3).

2.7 Research Issues

Derivation of Preferences from Social Networks. In the discussed prioritization scenarios, preference elicitation is still a manual process. Especially in contexts where companies have established a social network representing their user community, network contents, for example, in the form of tweets can be exploited to infer new requirements and preferences with regard to existing and future software features [26]. The automated integration of community preferences into requirements prioritization is still an open issue and extremely relevant for making related decision processes more community-oriented and efficient. Beyond automated preference integration, quality assurance for preferences is an extremely important issue. [34] show how a consequence-based evaluation of different choice alternatives can help to improve the overall quality of release planning decisions.

Avoidance of Decision Biases. Decision biases are related to shortcuts in decision making that can lead to sub-optimal decisions [35,36]. Being aware of such biases helps to improve the overall quality of decisions processes. An example of such a bias is *anchoring* where the item evaluations of one user that are already visible to other users who haven't evaluated the item up to now, can have an impact on the evaluation behavior of other users [33]. For an overview of decision biases in recommender systems we refer to [35]. Many of the existing biases reported in the psychological literature have not been evaluated up to now. This can be regarded as a major topic for future research.

Transparency of Decisions. In order to increase trust, decisions have to be made transparent. Transparency can be achieved on the basis of explanations, which help to understand the reasons for a recommended decision [30]. An important role of transparency is also related to the task of avoiding manipulations in decision making [37]. An example thereof is a situation where a user tries to adapt his/her rating in order to push his/her preferred alternatives (*push attack*). As discussed in Trang *et al.* [37], a very effective way of avoiding manipulations is to make the rating behavior of individual users more transparent, i.e., making their rating behavior visible to other users. A research issue in this context is to analyze in detail which degree of transparency of rating behavior best helps to counteract manipulations and which visualizations should be used to explain the current status of a decision process.

Prioritization and Decision Making in Open Source Environments. Open source development often takes place in the context of single user (contributor) decision making, i.e., contributors can individually and independently decide which requirement to implement next. Often, many new requirements are potential candidates and the analysis of these candidates is time-consuming. In this context, prioritization can help to automatically rank new requirements in a contributor-specific fashion and thus to significantly reduce related analysis efforts. An approach to support such prioritization scenarios in the Eclipse open source environment is reported in [29]. A research challenge in this context is to develop decision support approaches that do not only determine recommendations for individuals but also to figure out which prioritization helps to make the open source community as a whole more productive.

2.8 Conclusions

In this chapter, we provide an overview of prioritization scenarios that can be differentiated with regard to the degree of underlying requirement granularity and whether constraints are used to describe a prioritization task. These scenarios range from *early requirements engineering* (utility analysis of high-level requirements), *minimum viable product* (selection of features to be contained in a first version of a product), *basic release planning* (initial prioritization of requirements), to *integrated release planning* (detailed prioritization of requirements with regard to a predefined set of releases). To better show the application of related decision support techniques, we introduce a couple of prioritization examples. This chapter is concluded with an outline of open issues for future research.

Acknowledgment

The work presented in this chapter has been conducted within the scope of the Horizon2020 OpenReq Project (funded by the European Union).

References

[1] P. Achimugu, A. Selamat, R. Ibrahim and M. Mahrin, A systematic literature review of software requirements prioritization research, *Information and Software Technology* **56**, 6, pp. 568–585 (2014).
[2] M. R. Karim and G. Ruhe, Bi-objective genetic search for release planning in support of themes, in *Proceedings Symposium on Search Based Software Engineering*. Springer, pp. 123–137 (2014).
[3] L. Lehtola, M. Kauppinen and S. Kujala, Requirements prioritization challenges in practice, in *5th International Conference On Product Focused Software Process Improvement (PROFES)*. Kansai Science City, Japan, pp. 497–508 (2004).
[4] B. Mobasher and J. Cleland-Huang, Recommender Systems in Requirements Engineering, *AI Magazine* **32**, 3, pp. 81–89 (2011).
[5] M. Alenezi and S. Banitaan, Bug reports prioritization: Which features and classifier to use? in *12th International Conference on Machine Learning and Applications*, pp. 112–116 (2013).
[6] A. Perini, F. Ricca and A. Susi, Tool-supported requirements prioritization: Comparing the AHP and CBRank methods, *Information and Software Technology* **51**, 6, pp. 1021–1032 (2009).

[7] G. Ruhe, Software engineering decision support–a new paradigm for learning software organizations, in *International Workshop on Learning Software Organizations*. Springer, pp. 104–113 (2002).

[8] J. Xuan, H. Jiang, Z. Ren and W. Zou, Developer prioritization in bug repositories, in *34th International Conference on Software Engineering (ICSE)*. Zürich, Switzerland, pp. 25–35 (2012).

[9] D. Firesmith, Prioritizing Requirements, *Journal of Object Technology* **3**, 8, pp. 35–47 (2004).

[10] D. Ameller, C. Farre, X. Franch, D. Valerio and A. Cassarino, Towards continuous software release planning, in *24th IEEE International Conference on Software Analysis, Evoluation and Reengineering (SANER)*, pp. 402–406 (2017).

[11] G. Ruhe and M. Saliu, The art and science of software release planning, *IEEE Software* **22**, 6, pp. 47–53 (2005).

[12] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, London (1993).

[13] A. Felfernig and R. Burke, Constraint-based recommender systems: Technologies and research issues, in *ACM International Conference on Electronic Commerce (ICEC08)*. Innsbruck, Austria, pp. 17–26 (2008).

[14] M. Pazzani and D. Billsus, Learning and revising user profiles: The identification of interesting web sites, *Machine Learning* **27**, pp. 313–331 (1997).

[15] Y. Koren, R. Bell and C. Volinsky, Matrix factorization techniques for recommender systems, *IEEE Computer* **42**, 8, pp. 30–37 (2009).

[16] U. Junker, QuickXplain: Preferred Explanations and Relaxations for Over-Constrained Problems, in *19th National Conference on AI (AAAI04)*. San Jose, CA, pp. 167–172 (2004).

[17] A. Felfernig, M. Schubert and C. Zehentner, An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM)* **26**, 1, pp. 175–184 (2012).

[18] M. Nayebi and G. Ruhe, Analytical product release planning, in *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, pp. 550–580 (2015).

[19] G. Ruhe, *Product release planning: methods, tools and applications*. CRC Press (2010).

[20] F. Kifetew, A. Susi, D. Mutante, A. Perini, A. Siena and P. Busetta, Towards multi-decision-maker requirements prioritisation via multi-objective optimisation, in *Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering (CAiSE'17)*. Essen, Germany, pp. 137–144 (2017).

[21] G. Adomavicius, N. Manouselis and Y. Kwon, *Recommender Systems Handbook*, chap. Multi-Criteria Recommender Systems, 1st edn. Springer, pp. 769–803 (2010).

[22] S. Huang, Designing utility-based recommender systems for e-commerce: Evaluation of preference elicitation methods, *Electronic Commerce Research and Applications* **10**, 4, pp. 398–407 (2011).

[23] K. Wiegers, *Software Requirements*. Microsoft Press (2003).

[24] G. Ninaus, A. Felfernig, M. Stettinger, S. Reiterer, G. Leitner, L. Weninger and W. Schanil, Intellireq: Intelligent techniques for software requirements engineering, in *European Conference on Artificial Intelligence, Prestigious Applications of Intelligent Systems (PAIS)*, pp. 1161–1166 (2014).

[25] M. Atas, T. Tran, R. Samer, A. Felfernig and M. Stettinger, Liquid democracy in group-based configuration, in *Workshop on Configuration*. CEUR, Graz, Austria, pp. 93–98 (2018).

[26] G. Williams and A. Mahmoud, Mining twitter feeds for software user requirements, in *25th International Requirements Engineering Conference (RE)*. IEEE, Lisbon, Portugal, pp. 1–10 (2017).

[27] J. Dyer, Multi attribute utility theory, *International Series in Operations Research and Management Science* **78**, pp. 265–292 (1997).

[28] J. Karlsson and K. Ryan, A Cost-Value Approach for Prioritizing Requirements, *IEEE Software* **14**, 5, pp. 67–74 (1997).

[29] A. Felfernig, M. Stettinger, M. Atas, R. Samer, J. Nerlich, S. Scholz, J. Tiihonen and M. Raatikainen, Towards utility-based prioritization of requirements in open source environments, in *26th IEEE Conference on Requirements Engineering*. IEEE, Banff, Canada, pp. 406–411 (2018a).

[30] A. Felfernig, L. Boratto, M. Stettinger and M. Tkalcic, *Group Recommender Systems – An Introduction*. Springer (2018b).

[31] L. Lwakatare, T. Kilamo, T. Karvonen, T. Sauvola, V. Heikkiläc, J. Itkonen, P. Kuvaja, T. Mikkonen, M. Oivo and C. Lassenius, DevOps in practice: A multiple case study of five companies, *Information and Software Technology* **114**, pp. 217–230 (2019).

[32] A. Davis, The art of requirements triage, *IEEE Computer* **36**, 3, pp. 42–49 (2003).

[33] M. Stettinger, A. Felfernig, G. Leitner and S. Reiterer, Counteracting anchoring effects in group decision making, in *23rd Conference on User Modeling, Adaptation, and Personalization (UMAP'15)*, *LNCS*, Vol. 9146. Springer, Dublin, Ireland, pp. 118–130 (2015).

[34] M. Nayebi and G. Ruhe, Asymmetric release planning: Compromising satisfaction against dissatisfaction, *IEEE Transactions on Software Engineering* **45**, 9, pp. 839–857 (2018).

[35] A. Felfernig, Biases in decision making, in *Proceedings of the International Workshop on Decision Making and Recommender Systems 2014*, Vol. 1278. CEUR Proceedings, Bolzano, Italy, pp. 32–34 (2014).

[36] A. Felfernig, W. Maalej, M. Mandl, M. Schubert and F. Ricci, Recommendation and decision technologies for requirements engineering, in *ICSE 2010 Workshop on Recommender Systems in Software Engineering*. Cape Town, South Africa, pp. 1–5 (2010).

[37] T. Tran, A. Felfernig, V. Le, M. Atas, M. Stettinger and R. Samer, User interfaces for counteracting decision manipulation in group recommender systems, in *27th ACM Conference on User Modeling, Adaptation and Personalization (UMAP)*. Larnaca, Cyprus, pp. 93–98 (2019).