EXPECTATION

Personalized Explainable Artificial Intelligence for decentralized agents with heterogeneous knowledge

D3.1
Technical report detailing the developed models and data integration
[M25]

Version: 1.0
Last Update: 28/04/2023
Distribution Level: CO

The EXPECTATION Project Consortium groups Organizations involved:

Partner Name	Short	Country
University of Applied Sciences and Arts Western Switzerland	HES-SO	Switzerland
Özyeğin University	OZU	Turkiye
University of Bologna	UNIBO	Italy

Document Identity

Creation Date:	1 April 2023
Last Update:	1 May 2023

Revision History

Version	Edition	Author(s)	Date

Introduction

Recommender systems (RS) are computational systems aimed at supporting decision-making for human users in a given domain. Support is provided by means of recommendations (i.e., suggestions), automatically generated by the system, and consumed by the users. The recommendation generation procedure is commonly based on a corpus of knowledge consulted by the RS in response to a query or as a proactive behavior.

In explainable RS, users may challenge recommendations by asking for further explanations. An explanation consists of additional information to be presented to the user in order to motivate, clarify, or compare the recommendation process itself and its outcome. In turn, explanations are based on the same domain-specific corpus of knowledge used for recommendations, plus some meta-information about the recommendation process.

In personalizable RS, recommendations are customized to comply with the peculiarities (e.g., the profile information, or the preferences) of the particular user consuming them, as well as the context the user is situated into. For instance, the same user may query the system twice, at different moments in time and get different recommendations, possibly because:

- the temporal context has changed (e.g., morning vs. evening);
- the user profile has changed (e.g., different age / weight / BMI range);
- the user's preferences have changed;
- the corpus of knowledge has been updated.

In Expectation, we are interested in studying the support/impact of Multi-Agent Systems (MAS) in Personalizable and Explainable RS (PERS). There, the single RS is modelled as an autonomous computational agent encapsulating the strategy for interacting with the user—there including the criteria by which personalized recommendations and explanations are drawn for a particular user.

The value-added of the *multi*-agent part lies in the fact that, for any given domain, multiple agents may be present—potentially, one per user. Each agent would then focus on learning the peculiarities of the user assigned to it. However, since the domain is the same, all agents may (partially) share their corpi of knowledge. Sharing knowledge is expected to bring benefits in terms of the quality of recommendations and explanations.

Our envisioned approach relies on various sorts of data, to be integrated by agents in their recommendation- and explanation-generation processes. Accordingly, in this document, we focus on the problem of formalizing the data types and the corresponding data sources and provisioning processes. We also discuss how all such kinds of heterogeneous data are integrated by agents to derive personalized recommendations and explanations.

Nutritional PERS: Data Perspective

Regardless of implementation details, any personalizable and explainable RS would rely on data of various sorts. So far, we mentioned:

- the users' queries, as well as the corresponding recommendations and explanations;
- the domain-specific corpus of knowledge, from which both recommendations and explanations are constructed;
- users' profile data and user preferences, which should be taken into account to personalize any piece of information presented to the user;
- contextual data (i.e., meta-data enriching user queries with relevant information about the situation a query has been issued into).

In the particular case of Expectation, we focus on the nutritional domain. There, the RS aims at helping users follow their nutritional goals (e.g., losing, gaining, or maintaining weight) by providing recommendations – and, possibly, explanations – about what to eat when—either upon request or autonomously.

To do so, agents shall rely on a corpus of knowledge consisting of both

- common knowledge about food, including recipes, cuisines, and their ingredients, and the categorization and nutritional values of those ingredients;
- experts' knowledge about how to select food w.r.t. a given nutritional goal.

Furthermore, agents should also collect information about each particular user's profile (e.g., age, height, health conditions, dietary restrictions, etc.) and preferences (e.g., user –U likes/dislikes food – F). While profile information may be provided by users explicitly, preferences should be learned by the system dynamically.

Contextual information may describe for instance when (e.g., time of the day, or day in a dietary schedule), or where a given query has been issued.

Finally, queries may simply represent questions about what a given user *U* should eat in a given context *C*. Recommendations may consist of food name suggestions (possibly including information about how to cook or where to buy food). Explanations may provide details about how and why a particular suggestion has been proposed, as well as comparisons with other options that have not been proposed.

Accordingly, in this section, we delve into the details of data sources, schemas, provisioning, and workflow concerning the nutritional domain.

Common-Knowledge concerning Food

Common knowledge concerning food is quintessential to let any RS attain its basic goal, namely: recommending food. To serve this purpose, we assume the existence of an architectural component – i.e., the food ontology – containing information about all the possible meals, and their categorization, ingredients, recipes, and nutrients.

The food ontology should support several sorts of queries, including, but not limited to:

- selecting food by the ingredients/nutrients they are composed of;

- selecting food by cuisine type or by category;
- selecting food having (at least, at max, about) some amounts of ingredients/nutrients;
- selecting ingredients/nutrients corresponding to a given meal;
- clustering meals having similar amounts of nutrients.

Notably, queries of these sorts are required by the other components of the RS to construct food recommendations.

In compliance with semantic web principles, Figure 1 structures the food ontology schema.

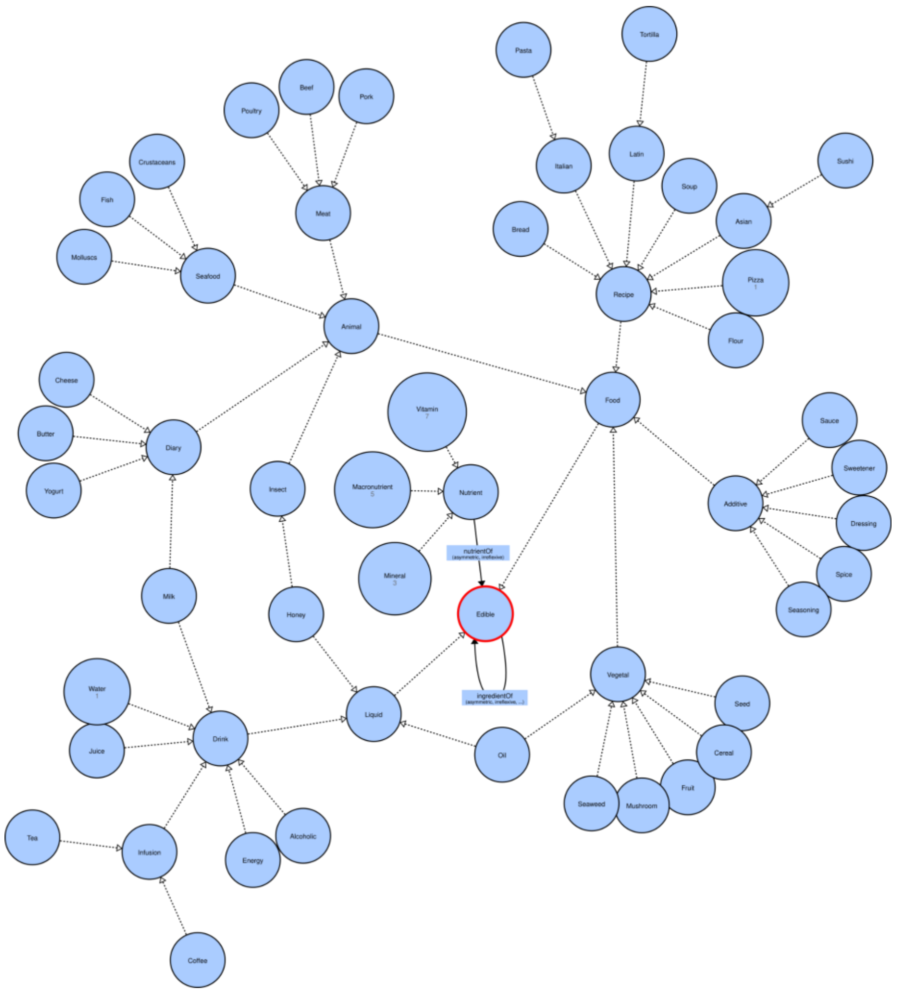


Figure 1 - Food ontology schema.

The ontology comprehends a hierarchy of classes representing categorization criteria for food or ingredients. The widest possible class is ‘Edible’, which represents anything that can be absorbed by the human body as a nutrient. Both ‘Food’ and ‘Liquid’ are sub-classes of ‘Edible’, aimed at representing solid and liquid edible stuff respectively.

Sub-classes of ‘Liquid’ may include beverages (‘Drink’ class), as well as ‘Honey’ or ‘Oil’—which are classes too, as there could be several particular sorts of oil or honey. Beverages, in turn, may include ‘Alcoholic’ drinks, various sorts of ‘Milk’ or ‘Vegetal Milk’, various flavors of ‘Water’, energy drinks (class ‘Energy’), or ‘Infusion’ beverages—such as ‘Tea’, ‘Coffee’, and so on.

Sub-classes of ‘Food’ may include both vegetal- and animal-derived food (classes ‘Vegetal’ and ‘Animal’ respectively), as well as ‘Additive’. Animal-derived food may include ‘Dairy’, ‘Meat’, and ‘Seafood’, which in turn may include specific sub-classes of products such as ‘Cheese’, ‘Beef’, or ‘Fish’. More detailed sub-classes have been defined as well, but they are not reported here for the sake of conciseness.

For some sorts of food items, the distinction between solid and liquid is not really relevant. This is the case for instance for the ‘Milk’ and ‘Honey’ classes. To model these situations, we exploit multiple inheritance. So, for example, both ‘Milk’ and ‘Honey’ are sub-classes of both ‘Diary’ and ‘Drink’.

Among the many direct sub-classes of food, one is devoted to keeping track of full recipes and their categorization. In fact, the class ‘Recipe’ contains various sorts of recipe names, categorized by cuisine. In particular ‘Recipe’ is a sub-class for the ‘Italian’ cuisine, for the ‘Asian’ cuisine, and so on and so forth. These sub-classes may have further sub-classes covering particular sorts of regional recipes.

The ontology also keeps track of foot nutrients (‘Nutrient’ class), which may be of three major sorts, namely:

- ‘Macronutrients’ (e.g., carbohydrate, fat, protein, etc.)
- ‘Mineral’, (e.g., calcium, iron, magnesium, etc.)
- ‘Vitamins’, (e.g., A, B6, B12, C, etc.)

Finally, the ontology tracks the properties of food as well. The ‘ingredientOf’ object property binds any two edible instances for which the first one is an ingredient of the second one. Conversely, the ‘nutrientOf’ object property binds nutrients with the edible instances containing them. In both cases, metadata concerning quantities may be provided by means of the ‘quantity’ annotation property. So, for instance, food X may be an ingredient of food Y, with quantity Q. It is worth mentioning that the ‘ingredientOf’ property is transitive, meaning that if some X is an ingredient of Y, and Y is an ingredient of Z, then X is also an (indirect) ingredient of Z.

Thanks to our food ontology, many queries involved in the recommendation and explanation procedures may be reduced to DL queries.

User Profiles in brief

Collecting rich user characterizations is quintessential to let any RS generate relevant, yet personalized, recommendations. Personalization, in particular, should span several domains, including cultural, ethical, and health-related conditions.

To serve this purpose, we assume the existence of an architectural component – i.e., the user ontology – containing information about the users and their profiles. Of course, the ontology is only conceptually unified. In practice, while different recommender agents may share the same classes and relations to represent user profiles, the actual data about specific real users is not centralized. Rather, we assume actual instances of the ontology are scattered among the many recommender agents, in such a way that each recommender agent only stores profile data about the user it is meant to provide recommendations to, and no data is shared with other users.

It is worth mentioning that the focus of this subsection is on user *profile* data – as opposed to user *preference* data. The distinction among such sorts of data is somewhat blurred. However, we rely on the following convention to discriminate. Profile data is, in principle, available ahead of time (w.r.t. when a given user starts to use the recommender system): the user is expected to be aware of its profile information, and that can be inserted into the system by means of some initial registration procedure. Conversely, preference data is very fine-grained, and the user may not be fully aware of it at sign-up time. Therefore, it is impractical to request preference data to the user during the initial registration phase. Rather, the recommender agent will need to grasp it later, by means of further interaction with the user.

Figure 2 provides an overview of the proposed ontology. The main covered/modelled areas are cultural and ethical factors, health and nutrition goals, needs, and preferences.

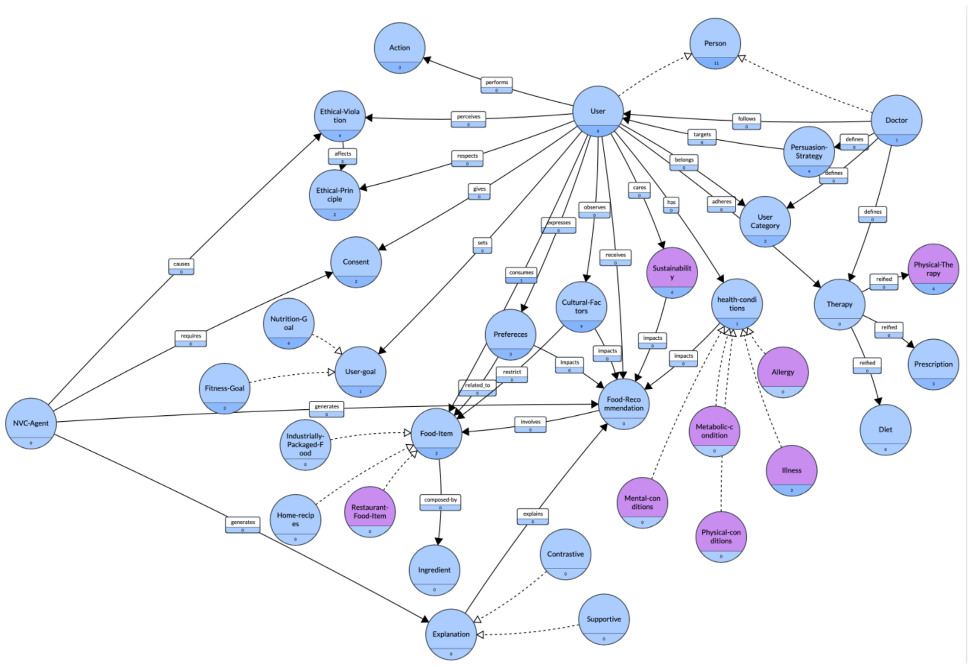


Figure 2 - User’s ontology for nutritional PERS.

Types of users

We consider two sorts of users, namely: “nutrition experts” (e.g., nutritionists and doctors), and “end users” (i.e., people willing to undertake behavioral change w.r.t. nutrition).

End users are the primary consumers of nutritional recommendations/explanations. They come with relevant (for the RS) profile and preference information, and they are ultimately benefitting from the usage of the system itself.

Nutrition experts are in charge of designing persuasion strategies and therapies for end users—there including prescriptions laying outside the scope of nutrition, yet affecting it. These prescriptions could be directed towards some specific individual user, or towards idealised user archetypes (e.g., "overweight male person, aged 50–60yo, willing to lose weight", and may involve dietary suggestions with a precise scheduling. When inserted into the system by some nutrition expert, such prescriptions should affect the recommendation/explanation strategies of all recommender agents corresponding to some end user matching the prescription.

Ontology description:

End-users and nutrition experts are represented by the *User* and *Doctor* classes. Both are subclasses of class *Person* representing people of any sort. Any person is characterized by anagraphic data – in the form of properties – such as personal/identification information (e.g., *id, name, password, email*, etc.) and demographic data (e.g., *age_range*, *county_of_origin*, *living country*, etc.).

Doctors may define user categories – i.e., custom groups of users – in order to better organize their nutritional plan. To serve this purpose, the ontology contains a class *User category*. Users may belong to user categories. Accordingly, *'belongs*' (resp. *'defines'*) is the relation biding *User* (resp. *Doctor*) to *User Category*.

Doctors may also assign *Users* with therapies, to which *Users* must *adhere*. This is relevant to our nutritional RS because recommendations should take into account the constraints posed by those therapies. Accordingly, therapies are instances of class *Therapy*. This class comes with three notable sub-classes, namely: *Physical-therapy*, *Prescription*, and *Diet*. *Physical-therapy* sub-class includes exercise recommendation from physiotherapist for fitness maintenance or for recovery from lesion. The sub-class *Prescription* is a medical document where a licensed practitioner orders medicines or therapy. The sub-class *Diet* is a meal plan that controls the intake of food and nutrients.

Notably, for any given user, our ontology may also memorize physical-therapies, prescriptions, or diets coming from experts which are not registered in the system. In fact, even if some of those elements are externally defined, it is necessary that both registered doctors and recommender agents are aware of them.

The ultimate purpose of each recommender agent is to help users in reaching their goal (class *User-Goal*). In practice, our ontology discriminates among *Nutrition-* and *Fitness-Goal*, the former focusing on what the user eats, and the latter on the amount of physical activity performed by the user.

To help the user reach their goal, recommender agents generate recommendations based on their *Health Condition*. This is a class with several relevant sub-classes, for specific sorts of conditions, namely *Allergy* (e.g., egg allergy), *Illness*, *Metabolic Condition* (e.g., Hunter syndrome)*, Physical Condition* (e.g., overweight), and *Mental condition* (e.g., Bulimia nervosa). Cultural and ethical factors may impact recommendations as well, therefore they come with ad-hoc classes (namely, *Cultural-* and *Ethical-Factor*). Notable examples of cultural factors are religion (e.g., kosher diet) or dietary habits (e.g., vegetarian). Conversely, one notable example of an ethical factor is *sustainability*, which may imply higher priority to local and seasonal food.

User Preferences and Tastes

We assume that users' tastes concerning individual edible instances are not explicitly represented, as the information is too fine-grained, and users are unlikely to input them manually. Rather, we require our recommender agents to be able to eventually learn users' preferences **sub-symbolically**.

Accordingly, we assume users' preferences are encoded into ML predictors. In other words, the recommender agent has a sub-symbolic component, which is capable of learning users' tastes adaptively, from historical data accumulated by interacting with the user. This component assumes data describing the recipes the user likes (or dislikes) are available in appreciable amounts.

Along this line, we let the triplet $\langle u, r, p \rangle$ denote a particular preference of user u. There, $r \in \mathfrak{R}$ is a recipe from the class of recipes (here denoted by $\mathfrak{R}$), and $p \in \mathbb{R}$ is an appreciation score, where positive values represent appreciation, negative values represent dislike, and 0 represent neutrality. In other words, the appreciation score encapsulates the user's opinion w.r.t. the recipe, which may be fuzzy.

Data representing the user's preferences is assumed to be collected by the recommender agent while interacting with the user, as part of its ordinary operation—possibly, via smart or wearable devices. In particular, data is exploited by the agent as the training set for its sub-symbolic component aimed at learning the user's tastes. Of course, in doing so, the learning algorithm may also access the ingredients- and nutrients-related information stored into the food ontology.

One key aspect of the sub-symbolic approach is that learning should be continual, in order to keep it adherent w.r.t. the user's preferences—which may evolve over time. Under such assumptions, users' preferences are modeled as a function

$$appreciation_u : \mathfrak{R} \rightarrow \mathbb{R}$$

aimed at predicting user u's appreciation score for any given recipe.

In practice, function $appreciation_u$ is approximated via a sub-symbolic predictor trained over the food ontology and user u's data, concerning previous appreciation information—either directly or indirectly provided by u to its corresponding recommender agent.

One important engineering aspect concerning the sub-symblic approximation procedure is how to model recipes sub-symbolically. Ideally, we want the sub-symbolic predictor to learn the *actual* preference of a user, exploiting the recipes they (dis)like as examples to be generalized. In this way, the predictor would be able to estimate the preference of the user, even for recipes that the user has never tasted! For this reason, we model the recipe as the set of ingredients it is composed of (and their quantities).

Accordingly, we denote by Γ the set of admissible ingredients, which we assume to be of finite cardinality. Under such hypothesis, we assume all recipes are represented as sets couples of the form $\{\langle\gamma_1, q_1\rangle, \ldots, \langle\gamma_n, q_n\rangle\}$, where each $\gamma_i \in \Gamma$ is an ingredient and $q \in \mathbb{R}$ is a quantity value. In other words, we consider $\mathfrak{R} \subseteq 2^{\Gamma\times\mathbb{R}}$.

At the coding level, such representation of recipes can be further reduced to vectors of real numbers. There, we could denote each recipe as a vector of real numbers of size $|\Gamma|$, where the i-th component denotes the quantity of the ingredient i in a given recipe—0, by default. Such quantities may be computed by propotionalizing the food ontology.

Experts' knowledge

Dietary prescriptions are structured representations of *what* a given user should eat, and *when*, in order (for the user) to achieve a particular *goal*. They are commonly produced by nutrition experts upon request, and structured around the particular physiological features of the user, other than the expert's background knowledge and experience.

For any given prescription, the 'what' part consists of particular recipes, ingredients, or nutrients the user should assume on a per-meal basis, along with the corresponding quantities. Conversely, the 'when' part indicates the moment of the day the meal should be consumed (e.g., breakfast, lunch, dinner, etc.). Finally, the goal is the long-term effect the expert is expecting to produce on the body of the user, under the assumption that the dietary prescriptions are followed accordingly. The goal should reflect the user's request, but it does not need to be explicitly represented in the prescription.

Considering all the ontological information described so far, prescriptions can be modelled as a relation among users, edibles, time slots of the week, and quantities. We call this relation "*should_eat*". More formally, we let the quartet $\langle u, e, t, q\rangle$ denote a particular prescription for user u, which should eat q units of e at time t. In the general case, $q \in \mathbb{R}_{>\mathbf{0}}$ denotes a positive measure of the quantity of edible thing to be consumed—this may be weight, volume, or integer amount, depending on the nature of the edible thing. Conversely, $t \in \mathbb{N}$ denotes a particular moment in time corresponding to a scheduled meal, assuming a *discrete* notion of time. This could be for instance a number from 1 to 21, in case the user is willing to take 3 meals per day (breakfast, lunch, dinner) on a weekly basis (7×3).

It is worth mentioning that OWL ontologies do not support quaternary relations. To overcome this limitation, we implement the *should_eat* relation as a binary one (among *User* and *Edible*), and we require each instance of such relation to be **annotated** with a number (representing q) and an instance of *Meal-Time* (representing t)—that is, the class containing admissible time-slots for meals (e.g.,"monday at lunch").

In the real world, dietary prescriptions usually come in a quasi-natural language form or in tabular form. In this case, each cell represents a particular moment of the week, and the nutrients/ingredients/recipes (and their corresponding quantities) therein suggested by the expert. Therefore, each cell corresponds to one possible instance of the aforementioned "*should_eat*" relation. When this is the case, we assume the existence of a (semi-)automated procedure for convering tables into machine-interpretable data, compliant with our ontology.

Queries, Recommendations, and Explanations

Roughly speaking, the interaction among a recommender agent and its corresponding user can be summarized as follows:

1. the user sends a query towards the recommender agent (in our case, asking for food recommendations), or schedules the recommendation for a later moment in time;
2. the agent computes a recommendation and sends it back;
3. the user asks for an explanation concerning the recommendation(s) the recommendations they received so far;
4. the agent provides such explanation(s).

While the details of this interaction are discussed in the next section, here we delve into the details of how queries, recommendations, and explanations are actually represented.

Queries

Queries are simple requests of recommendations about what to eat. They may carry additional constraints, possibly reifying some contingent desire or need of the user. For instance, the query "can you recommend some italian recipe?" is explicitly constraining the search space to Italian recipes only. However, queries also carry metadata aimed at *contextualizing* them. To support personalization of recommendation, metadata should at least include information about *who* issued the query, and *when*—other than, possibly, *where*.

So, each query should carry metadata including (i) the identifier of the user issuing the request, (ii) an indication of the moment in time when the recommendation should be consumed – e.g., a timestamp, or an ordinal value according to the schema discussed in the previous section –, and, possibly, (iii) a indication of the position where the user shall consume the recommendation.

In this way, by means of the user identifier carried by some query's metadata, the recommender system may tailor answers on the particular user issuing the request, customizing them on the basis of the profile and preference information available for that user. Recommendations (and therefore explanations) may also be enhanced by means of the temporal and spatial indication carried by metadata. Temporal indication may be for instance combined with cultural conventions about time and nutrition (e.g., some people prefer a salty breakfast over a sweet breakfast, or vice versa). Similarly, spacial indications may be taken into account by promoting recommendations which can be locally bought.

Recommendations

Recommendations are responses to queries. In our nutritional domain, recommendations aim at suggesting edible things to consume. Conceptually, they are simple data structures, i.e.,sets of edible things being recommended. However, theory and practice greatly differ in this case.

In theory, one recommendation is simply a set of identifiers, each one identifying one particular item in our food ontology—like, for instance, URLs. However, in practice, the **presentation** of the recommendation plays a crucial role in the user experience of human beings. In other words, when showing the recommendation to the user, the mere URL may be poorly effective. The recommendation should rather be presented in a catchy way, possibly including concise names, pictures, a summary of nutritional values, or steps to reproduce, etc.

Explanations

Explanations are details supporting some recommendations. They may be of two broad types, namely:

1. ordinary explanations, which aim at answering the question "why did you recommend me this?";
2. contrastive explanations, which aim at answering the question "why did you not recommend me that instead?".

Ordinary explanations are "absolute", in the sense that they require the recommender agent to generate further information motivating some prior recommendation. An explanation of this sort may be attained by extracting the decision process which led the agent to propose that recommendation and by presenting such information accordingly to the user. Hence, an ordinary explanation is an intelligible representation of some decision process whose outcome is affecting the user.

Conversely, contrastive explanations are "relative" to an expectation of the explainee, in the sense that they require the recommender agent to compare its recommendation with the one the user was expecting. An explanation of this sort may be attained by describing the main differences among the two recommendations – in case both are adequate –, or by motivating why one of the two is not adequate.

Ordinary Explanations

In order to generate such explanations, we borrow the concept of post-hoc explanation generation from the xAI literature in the form shown in the figure below. The explanations are "post-hoc" meaning after the recommendation has been made, a simpler model that is inherently explainable and attempts to reason on the outcomes and surrogated model of the black box.

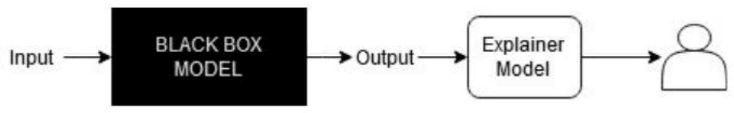


Figure 3 - black-back schematization.

For the explanation generation model, we have utilized decision trees in two manners in which we label the data. First, the item-based trees are trees generated from user preferences whereas the user-based trees are generated from the opinions of the other users. When we employ the user-based explanation generation method, the decision tree is constructed from historical data in which recipes are labelled **with all users' decisions** (i.e., accept or reject). On the other hand, the item-based explanation generation approach utilizes the decision tree constructed from a set of recipes labelled according to **the current user's constraints and feedback**. For that tree, filtered and low-scoring recipes are negatively labelled (-1), recipes that align with the user's constraints are positively labelled (+1) and the rest is labelled neutrally (0). After sorting features with respect to their importance, we chose **three** of them to generate a set of explanations for the given recipe.

After our feature selection, we follow a grammar-based structure to generate sentences from our selected features. For instance, if the fiber feature was considered important, then a sentence generated from that feature would be: *"This recipe is good in fibers"*.

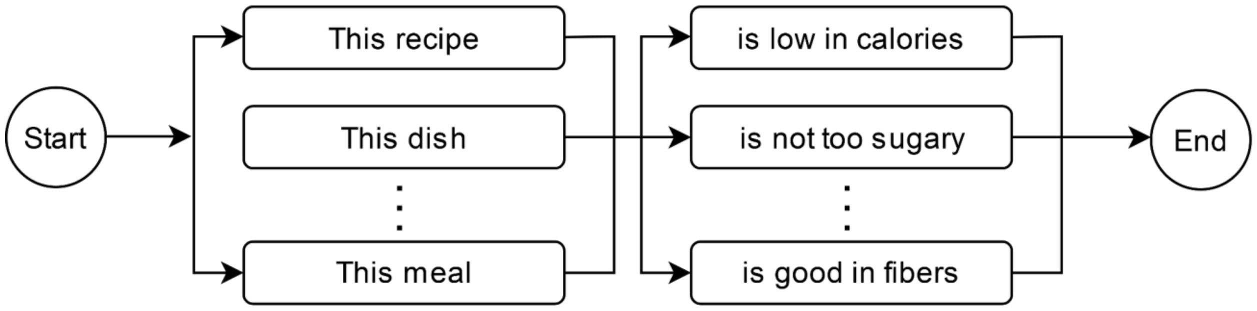


Figure 4 - Grammar structure of the ordinary explanations

Contrastive Explanations

For the contrastive explanations, we follow a different strategy. In order to generate a contrastive explanation, we must also choose a contrasting recipe (e.g., a recipe that is unfavorable but comparable to a given recommendation). To select such a recipe, we select from the pool of undesired recipes and compare the features of our recipe and the selected contrastive recipe. The features that highlight the positive effects of our recipe are selected to go through a different grammar structure.

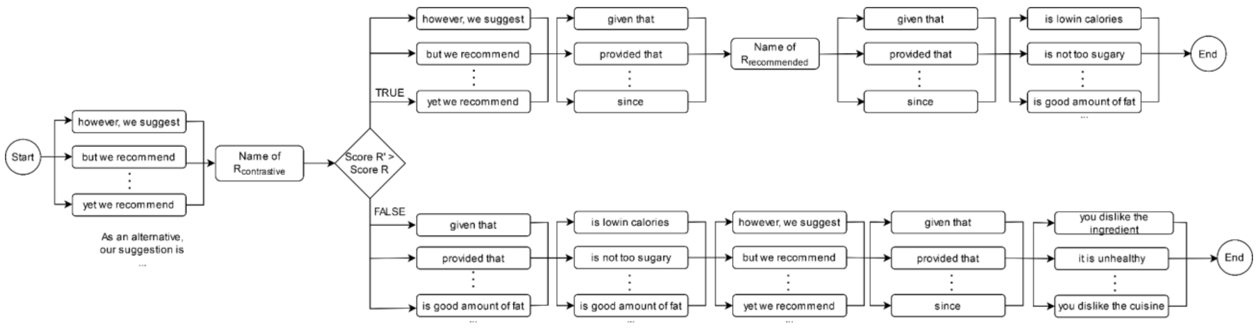


Figure 5 - Grammar structure of the contrastive explanations

PERS: Interaction Perspective

The interaction between one recommender agent and its corresponding user is regulated by a precise protocol, discussed in this section.

The protocol assumes that the user is in charge of initiating the interaction. Hence, the agent waits for the user to trigger a query. When receiving a query, the agent should respond by producing a recommendation.

While computing the recommendation, the agent may leverage any information available to it at that moment, including the user's profile, the history of previous interactions, and – possibly – aggregated information about other users. Furthermore, it may take advantage of both symbolic AI reasoning facilities, and machine learning predictors.

In response to a recommendation, the user may either simply accept/discard the recommendation, or ask for explanations.

The explanation phase may involve several rounds of interaction, where the user may either ask for further details or request comparisons; and the agent attempts to provide all such kinds of information. Eventually, enlightened by the explanation process, the user may either accept or reject the recommendation. In both cases, the agent may consider the acceptance/rejection of its recommendation – as well as the amount of explanatory information provided required by the user to reach a decision – as feedback for future recommendations. In the particular case of a rejection, the agent may also be interested in the reason for the rejection, so as to improve its recommendation and explanation strategy.

Notably, explanations are always (i) provided upon request, (ii) related to the recommendation, and (iii) directed towards the user. Accordingly, our protocol supports both types of explanations, and it lets the user decide which type of explanation to request. Of course, the exchanged messages may be different depending on the type of explanation requested.

Here we propose an abstract formulation of the protocol which is agnostic w.r.t. the particular way in which the recommendation and explanation are represented and computed. In other words, we only focus on the messages exchanged among explainers and explanees, what information they should carry, and in which order they should be exchanged. Accordingly, the protocols rely on 13 types of messages, which may carry data fields of 5 different types, to be exchanged among agents playing 2 possible roles.

Roles are of course explainee (a.k.a. user) and explainer (a.k.a. agent). The explainee initiates the protocol, while the explainer waits for the protocol to be started by the explainee.

We also identify 5 data types that represent the potential payload that agents may exchange during the protocol. As far as the abstract formulation of our protocol is concerned, we do not constrain the shape/structure of these types, but we simply assume they exist. In this way, implementers of the protocol will be free to define their own specifications for these types, tailoring them to their particular application domain. In particular, the data types are:

- **Queries** (denoted by *Q*), i.e., recommendation requests concerning a given topic, issued by the explainee when initiating the protocol;
- **Recommendations** (denoted by *R*,*R'*), i.e., responses to queries, issued by the explainer;
- **Explanations** (denoted by *E*,*E'*), i.e., chunks of explanatory information issued by the explainer to clarify their recommendation;
- **Features** (denoted by *F*), i.e., aspects of the user which are relevant which justify some recommendation rejection, which the explainer should memorize and take into account in future interactions;
- **Motivations** (denoted by *M*), i.e., reasons for the rejection of a recommendation, which may affect how the agent reacts to a rejection.

Finally, we identify 13 types of messages, which are exchanged among agents playing the explainee and explainer roles. We denote messages as named records of the form: `Name(Payload)`, where `Name` represents the type of the message and `Payload` represents the data carried by the message—which consists of instances of the aforementioned data types. Payloads consist of ordered tuples of data types, where items suffixed by a question mark are optional. Accordingly, message types are:

- `Query(Q)` is the message issued by the explainee to initiate the protocol: it carries a recommendation request `Q`;
- `Recommendation(Q,R)` is the message issued by the explainer in response to a query: it carries the query `Q` and the corresponding recommendation `R` computed by the explainer;
- `Why(Q,R)` is the message issued by the explainee to request an explanation of a recommendation: it carries the original query `Q` and the recommendation `R`;
- `WhyNot(Q,R,R')` is the message issued by the explainee to request a contrastive explanation of a recommendation: it carries the original query `Q`, the recommendation `R`, and a second recommendation `R'`, which the explainee wants the explainer to contrast with `R`;

- `Accept(Q,R,E?)` is the message issued by the explainee to accept a recommendation: it carries the original query `Q`, the recommendation `R`, and optionally the explanation `E` provided by the explainer;
- `Collision(Q,R, F,E?)` is the message issued by the explainee to notify the explainer that the provided recommendation is colliding with some personal feature/preference of theirs: it carries the original query `Q`, the recommendation `R`, a description of the feature `F`, and optionally the explanation `E` provided by the explainer;
- `Disapprove(Q,R,M,E?)` is the message issued by the explainee to notify the explainer that the provided recommendation is not acceptable for some reason: it carries the original query `Q`, the recommendation `R`, a description of the reason `M`, and optionally the explanation `E` provided by the explainer;
- `Details(Q,R,E)` is the message issued by the explainer to provide more details about a recommendation: it carries the original query `Q`, the recommendation `R`, and the explanation `E`;
- `Comparison(Q,R,R′,E)` is the message issued by the explainer to provide a contrastive explanation of a recommendation, in the case the one recommendation proposed by the explainee is admissible as well: it carries the original query `Q`, the recommendation `R` computed by the explainer and the one `R′` proposed by the explainee, and an explanation `E` comparing the two;
- `Invalid(Q,R′,E)` is the message issued by the explainer to notify the explainee that the proposed recommendation is invalid: it carries the original query `Q`, the proposed (and invalid) recommendation `R′`, and an explanation `E` motivating the invalidity;
- `Unclear(Q,R,E)` is the message issued by the explainee to notify the explainer that the provided explanation is unclear: it carries the original query `Q`, the recommendation R, and the provided (and unclear) explanation `E`;
- `Prefer(Q,R,R′)` is the message issued by the explainee to notify the explainer that they prefer a different recommendation: it carries the original query `Q`, the recommendation `R` proposed by the explainer, and the preferred recommendation `R′` proposed by the explainee;
- `Override(Q,R,R′)` is the message issued by the explainee to notify the explainer that want to force the decision to some recommendation which is considered invalid by the explainer: it carries the original query `Q`, the recommendation `R` proposed by the explainer, and the forced recommendation `R′` proposed by the explainee.

The following diagram summarizes the message flow described above:

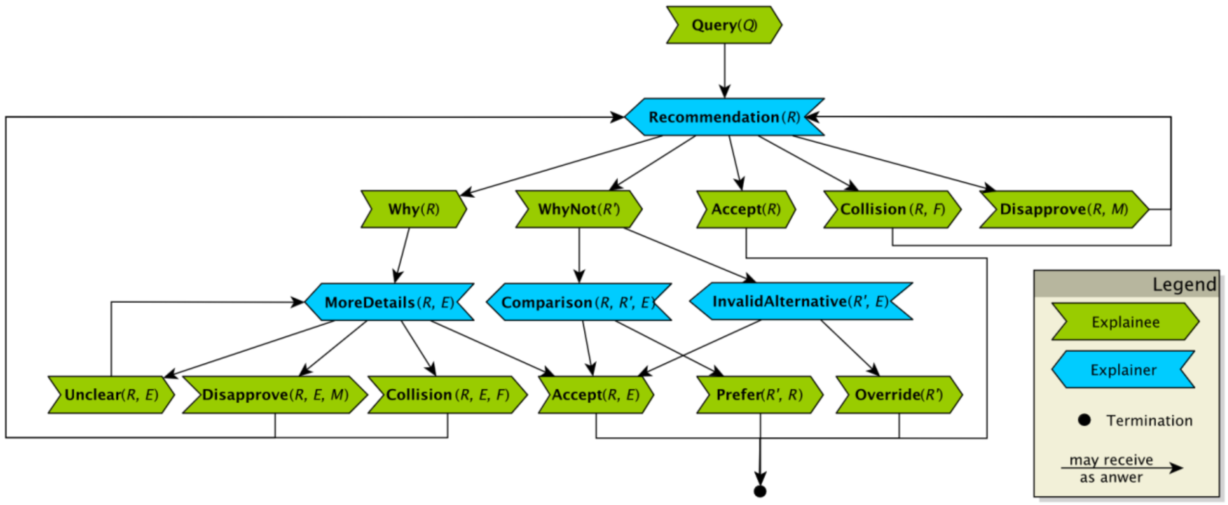


Figure 6 - Message communication diagram between an explainer agent (blue boxes) and an explainee (green boxes). Each box represents a message. Each message is connected to the ones it can receive as reply.

Notably, messages are designed by keeping the representational state transfer (ReST) architectural style into account—meaning that each message carries all the information needed to process it.

The message communication diagram above depicts not only the messages exchanged by the explainee and explainer, but also the admissbile request–response patterns which the protocol allows. There, a more detailed view of the message flow is provided, which we briefly summarize in the following. The explanation-based recommendation protocol consists in the following phases (depth-first traversal of the diagram above):

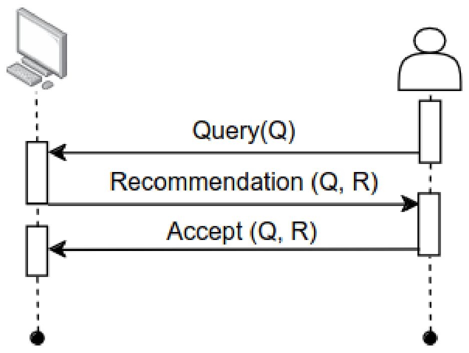
1. the explainee initiates the protocol, by issuing a message `Query(Q);`
2. the explainer provides a message `Recommendation(Q,R)` in return;
3. the explainee may now:
   a. accept the recommendation, by answering `Accept(Q,R)`, hence terminating the protocol;
   b. reject the recommendation because of `M`, by answering `Disapprove(Q,R,M)`; or signal it as colliding with `F`, by answering `Collision(Q,R,F)`. In this case, the explainer should propose another recommendation (go to 2.);
   c. ask for ordinary explanations, by answering `Why(Q,R)`. In this case, the explainer should propose an explanation, by answering `Details(R,E)`. The explainee may now:
      i. accept, reject, or signal `R` in light of `E`, by answering `Accept(Q,R,E)`, `Disapprove(Q,R,M,E)`, or `Collision(Q,R,F,E)`, respectively, with outcomes similar to cases 3.a. and 3.b.;
      ii. ask for a better explanation via `Unclear(Q,R,E)` (go to 3.c.).
   d. ask for contrastive explanations motivating why not `R'`, by answering `WhyNot(Q,R,R')`. The explainer may now:

i. explain the difference `E` among `R` and `R′`, if `R′` is admissible w.r.t. Its current knowledge base, by answering `Comparison(Q,R,R′,E)`. Now, the explainee may either (in both cases, the protocol terminates):
   1. accept `R`, via `Accept(Q,R,E)`, or
   2. state that they prefer `R′`, via `Prefer(Q,R,R′)`.

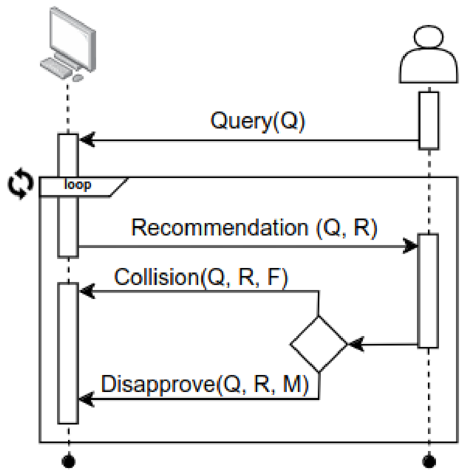
ii. explain that `R′` is not an admissible recommendation because of `E`, by answering `Invalid(Q,R′,E)`. At this point, the explainee may either (in both cases, the protocol terminates):
   1. accept R, via `Accept(Q,R,E)`, or
   2. override the explainer's decision, by stating that they prefer `R′`, via `Override(Q,R,R′)`—hence forcing the explainer to update their own knowledge base accordingly

The protocol is general enough to cover multiple relevant situations, corresponding to different needs/desires of the users. For instance, users may: (i) simply want a recommendation; (ii) want the recommendation to be explained; (iii) want more details for a given explanation; (iv) want to simulate other possible recommendations; (v) provide positive or negative feedback about recommendations or explanations.

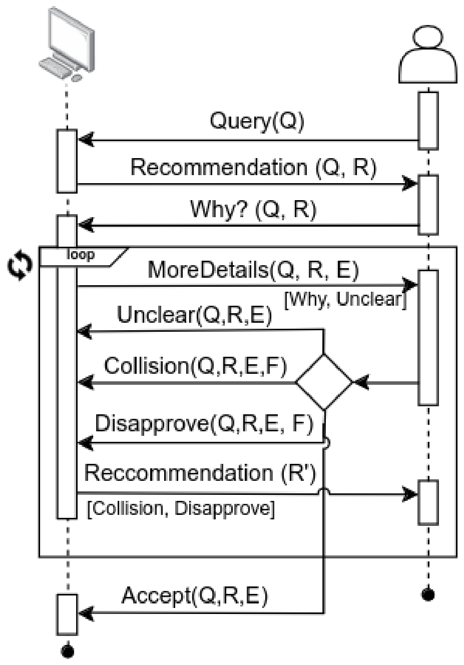
All such situations correspond to relevant usage scenarios of the protocol. These are briefly summarized in the figure below:



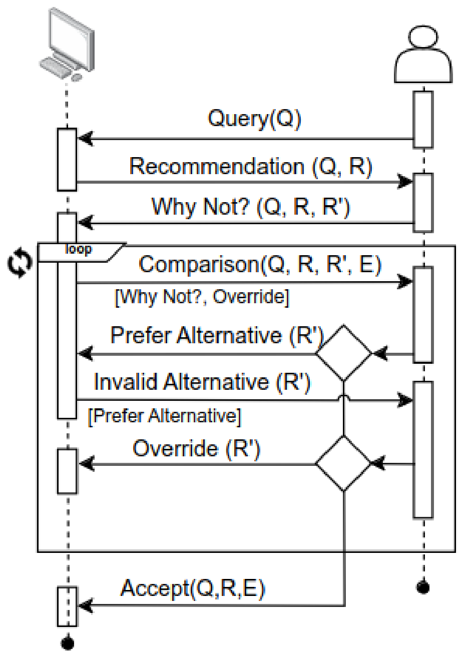
(a) Quick accept: the user accepts the recommendation without asking for explanations.



(b) Quick retry: the user rejects the recommendation without asking for explanations. Another recommendation is proposed, accordingly.



(c) Ordinary explanation loop: the user asks 'why' after a recommendation, and then agent answers with further details. The request for details may be repeated several times.



(d) Contrastive explanation loop: the user asks 'why not' another recommendation. The agent may then explain why the other recommendation is acceptable or invalid. The user may either accept the original recommendation or prefer their own.

Figure 7 - .

Data Representation and integration of Users' profiles/preferences into PERS

Our architecture leverages upon Horn clauses to represent users' preferences and experts' prescriptions. In fact, as we show in the remainder of this section, Horn logic lets us simply and clearly express dietary prescriptions, while retaining acceptable computational tractability features.
Such formulae may be written by *humans* and exploited by nutritional RS to suggest actual recipes to eat, or, vice versa, they may be generated by some algorithm, and understood by human beings as computer-generated prescriptions.
More precisely, experts prescriptions at time *t* consist of a set of Horn clauses defining the should eat/1 predicate. Such predicate intensively describes what recipes the user should eat at time *t*, by describing admissible (or forbidden) ingredients / nutrients. This is performed via two more predicates – namely, has/2 and has no/2 –, which assert what ingredients / nutrients the suggested recipe should or should not be composed by. Groups of ingredients / nutrients may be defined as well, via unary predicates defined by ad-hoc clauses—e.g., vegetable/1.

Users' preferences can be represented in clausal form as well. In that case, they consist of sets of Horn clauses defining the likes/1 predicate. Similar to prescriptions, such definitions may exploit the predicates has/2 and has no/2 too, as well as any other custom predicate defining groups of ingredients / nutrients.

Our architecture requires that both users' preferences and experts' prescriptions are available as sets of Horn clauses. Under such an assumption, it can construct recommendations via logic resolution.
As far as prescriptions are concerned, we assume that experts can produce them in clausal form, directly—or, at least, in forms which can be automatically converted into sets of Horn clauses. This assumption is easily met by the current practice of providing prescriptions as timetable of suggested recipes.
Conversely, as far as preferences are concerned, the clausal form requirement is clearly conflicting with eq. (2), where users' preferences are modelled as trained sub-symbolic predictors. The sub-symbolic representation is adequate, as it enables learning users' preferences from data, and adapting to their change over time. However, such form prevents the direct exploitation of logic resolution as the means to construct recommendation.
Accordingly, to fill the gap, we choose to bring users' preferences in clausal form, algorithmically. To serve this purpose, our architecture leverages upon a SKE step [1], which is in charge of extracting symbolic knowledge – in clausal form – out of the sub-symbolic predictor, which has been trained to predict users' preferences. Again, we do not impose any particular SKE algorithm – meaning that implementers are free to choose the extraction algorithm which is most adequate for their needs –, but we do require the extraction step and we require it to output Horn clauses.

Explanation Generation Models

Sub-symbolic predictors are able to identify hidden patterns in large amounts of data and generalize them to make decisions. Despite their high performance in several complex applications, sub-symbolic predictors are usually opaque, making it difficult to understand the reasons underneath their decision-making process. To understand the predictors' decision process and to make them trustworthy, it is necessary to provide them with knowledge-based explanatory mechanisms. Explainable Artificial Intelligence (XAI) is a field of artificial intelligence that aims to explain the predictors' decision process to make them transparent and trustworthy. To generate explanations and extract knowledge from sub-symbolic predictors, we have developed two main tools and platforms:

1. **PSyKE:** PSyKE (Platform for Symbolic Knowledge Extraction) is a knowledge extraction library which allows knowledge extraction from various sub-symbolic predictors in tasks such as classification and regression employing tabular data. PSyKE employs explainable-by-design models (e.g., decision trees) to approximate and describe the predictors' global behavior through first logic rules sets [1]. Rule sets generated by PSyKE are expressed in terms of the input features, maintaining their names, ranges and semantics. PSyKE employs tabular data (structured data in tabular form, where the columns are features and rows are data samples.)
2. **DEXiRE:** Deep Explanation and Rule Extraction (DEXiRE) is a knowledge extraction tool that explains the internal decision process on deep learning (DL) predictors trained on tabular data. DEXiRE's pipeline is shown in Figure 8 and starts with the predictors' prediction extraction. Subsequently, DEXiRE binarises neuron activations in the predictor's hidden layers. Then binary activation patterns from each hidden layer are generated employing the train set, and most frequently, patterns per class are identified and transformed in Boolean rule sets (intermediate rule sets). Later intermediate rule sets are pruned to reduce their complexity and expressed in terms of input features. Finally, intermediate rule sets are merged to produce the final rule set that describes the predictors' global behavior in terms of input features. Currently, DEXiRE is suitable for knowledge extraction on classification tasks with tabular data (e.g., users' preferences, food ingredients, snack packaging, and nutritional information.) [2].

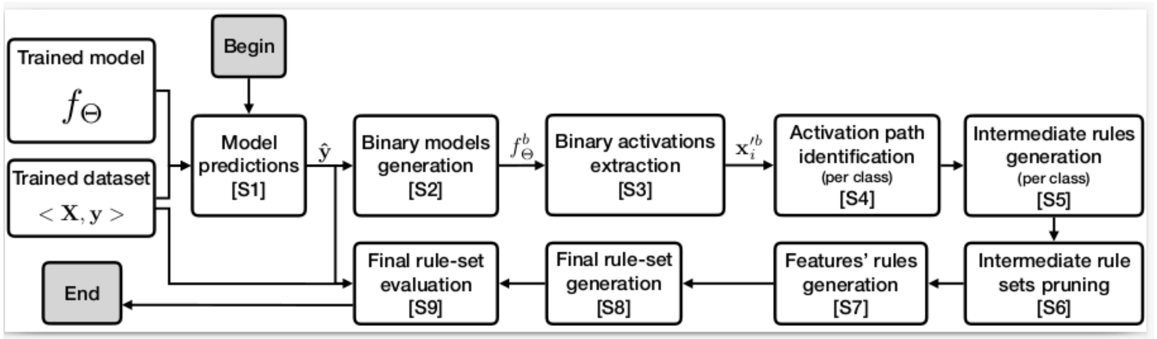

Trained model
f_Θ
Begin
Trained dataset
$<\mathbf{X}, \mathbf{y}>$
Model predictions [S1]
$\hat{\mathbf{y}}$
Binary models generation [S2]
f^b_Θ
Binary activations extraction [S3]
$\mathbf{x}'^b_i$
Activation path identification (per class) [S4]
Intermediate rules generation (per class) [S5]
End
Final rule-set evaluation [S9]
Final rule-set generation [S8]
Features' rules generation [S7]
Intermediate rule sets pruning [S6]

Figure 8: DEXiRE's pipeline [2].

EREBOTS: An agent-based PERS framework

EREBOTS is a multi-agent platform that enables the configuration and deployment of personalized chatbots to support users in multi-topic (and multi-campaign) behavioral change and health promotion programs. For example, EREBOTS'S conversational agent can coach individuals struggling with chronic diseases, addictions, and other health issues [4].
To provide its coaching services, EREBOTS uses and produces a large amount of data that is stored in two databases according to data privacy requirements:

- **MongoDB:** MongoDB is a high-efficient, fault-tolerant No-SQL database which stores non-sensitive data such as recipes, ingredients, user feedback to the application, agents' states, and current tokens to access chat front-end like Telegram tokens.
- **Pryv:** Pryv is a GDPR-compliant No-SQL database that ensures user privacy and security through dynamic consent, allowing the user to grant or remove authorization to use their data for different applications. Pryv stores all data subject to privacy or ethical criteria, including users' profiles, preferences, cultural and religious factors, allergies, and medical or psychological conditions. Additionally, Pryv stores the conversational history between the virtual coach and the user, the conversations between virtual coaches and other agents, and other service messages to synchronize agent behaviors.

As illustrated in Figure 9, all the databases (MongoDB and Pryv) employed by the EREBOTS platform are deployed in independent containers, isolating them from the rest of the components enabling greater flexibility, resilience, and fault tolerance of the system.

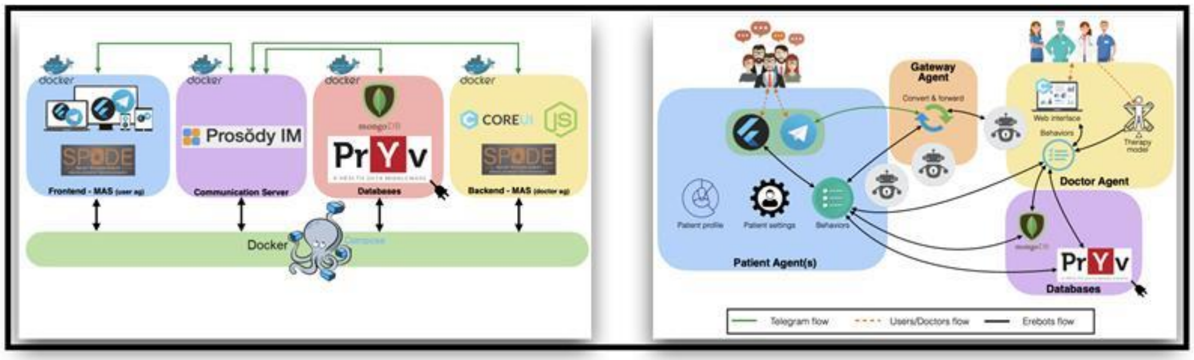


Figure 9 - EREBOTS architecture and component-interactions diagram.

Figure 10 indicates the architectural design and data flow in the EREBOTS 2.0 platform. The blue box shows the patient agent (PA), a personalized agent that implements three finite state machines (FSM), one thematic which constitutes the behavior of the Virtual Nutrition Coach (NVC). One persuasive FSM that implements and executes negotiation and behavioral change strategies to ensure the continuity of treatment and sustainability of results. Additionally, the PA implements a functional FSM that receives and redirects the messages to the correct FSM, updates the agent's state, and initializes additional behaviors. PA employs the user's profile and patient's settings to provide personalized recommendations and guidance.

Each user is associated with a unique PA agent (personal coach), and only their associated PA can access their private data stored in the Pryv database once the user has consented. This ensures controlled access to user data. The Gateway Agent (GA) adapts PA messages to the front end (e.g., HemerApp or Telegram). GA does not store or access data.

Figure 10 shows that Doctor Agent is a *special type* of agent intended to allow doctors (i.e,. nutritionists), to manage and monitor the users and their trends. Nutritionits can (via the Doctor Agent web interface) create customized or general plans to be executed by PA agents. Additionally, Doctor Agent has access to the statistics gathered by PA agents.

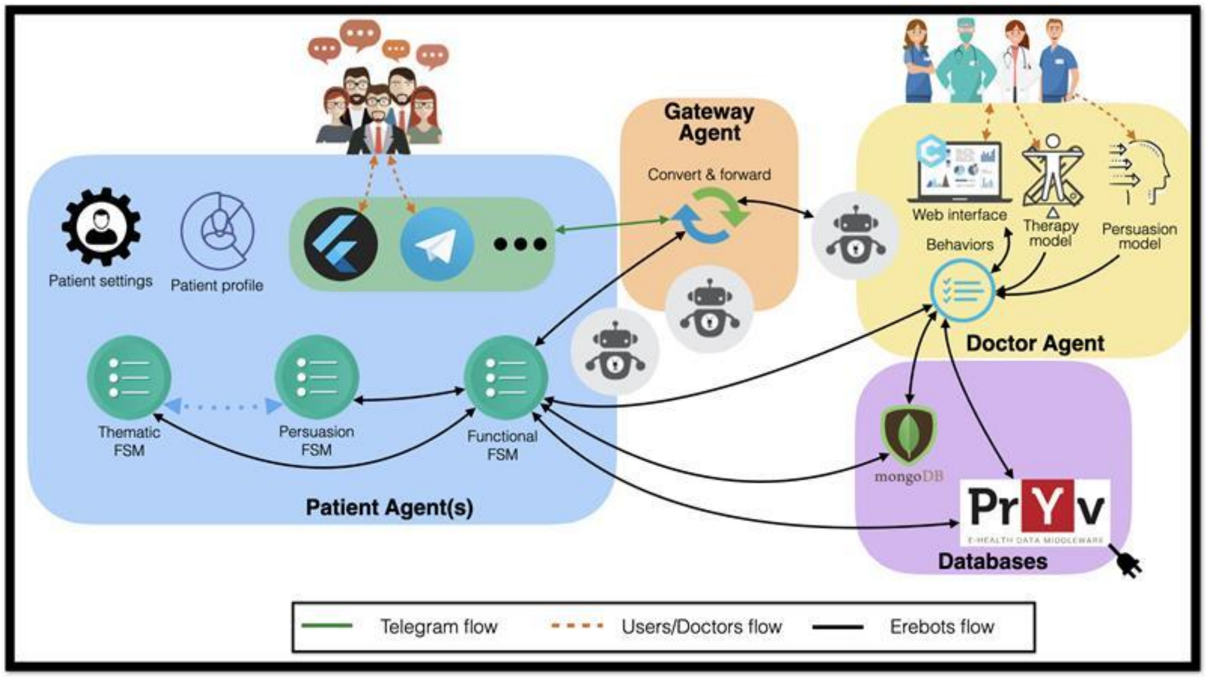


Figure 10 - Architectural design EREBOTS 2.0.

References

1. Sabbatini, F., Ciatto, G., Calegari, R., & Omicini, A. (2021, September). On the Design of PSyKE: A Platform for Symbolic Knowledge Extraction. In *WOA* (pp. 29-48).
2. Contreras, V., Marini, N., Fanda, L., Manzo, G., Mualla, Y., Calbimonte, J. P., ... & Calvaresi, D. (2022). A DEXiRE for Extracting Propositional Rules from Neural Networks via Binarization. *Electronics*, *11*(24), 4171.
3. Palanca, J., Terrasa, A., Julian, V., & Carrascosa, C. (2020). Spade 3: Supporting the new generation of multi-agent systems. *IEEE Access*, *8*, 182537-182549.
4. Calvaresi, D., Calbimonte, J. P., Siboni, E., Eggenschwiler, S., Manzo, G., Hilfiker, R., & Schumacher, M. (2021). EREBOTS: Privacy-compliant agent-based platform for multi-scenario personalized health-assistant chatbots. *Electronics*, *10*(6), 666.