A General-Purpose Protocol for Multi-agent Based Explanations

Giovanni Ciatto[1(✉)], Matteo Magnini[1], Berk Buzcu[2], Reyhan Aydoğan[2,3], and Andrea Omicini[1]

[1] Department of Computer Science and Engineering (DISI), Alma Mater Studiorum – Università di Bologna, via dell'Università 50, 47522 Cesena, FC, Italy
{giovanni.ciatto,matteo.magnini,andrea.omicini}@unibo.it

[2] Department of Computer Science, Özyeğin University, Nisantepe Mah. Orman Sok. No: 34-36 Alemdağ, Çekmeköy, 34794 Istanbul, Turkey
berk.buzcu@ozu.edu.tr

[3] Interactive Intelligence, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
reyhan.aydogan@ozyegin.edu.tr

Abstract. Building on prior works on explanation negotiation protocols, this paper proposes a general-purpose protocol for multi-agent systems where recommender agents may need to provide explanations for their recommendations. The protocol specifies the roles and responsibilities of the explainee and the explainer agent and the types of information that should be exchanged between them to ensure a clear and effective explanation. However, it does not prescribe any particular sort of recommendation or explanation, hence remaining *agnostic* w.r.t. such notions. Novelty lays in the extended support for both ordinary and contrastive explanations, as well as for the situation where no explanation is needed as none is requested by the explainee.

Accordingly, we formally present and analyse the protocol, motivating its design and discussing its generality. We also discuss the reification of the protocol into a re-usable software library, namely PyXMas, which is meant to support developers willing to build explainable MAS leveraging our protocol. Finally, we discuss how custom notions of recommendation and explanation can be easily plugged into PyXMas.

Keywords: XAI · recommender systems · multi-agent systems · explanation protocols · Spade · PyXMas

1 Introduction

Explainable AI (XAI) is an area of research aimed at developing AI systems that can provide understandable explanations of their decisions or behaviours to humans [11]. The need for XAI arises from the fact that many modern AI systems, particularly those based on deep learning and other forms of machine learning, are often seen as "black boxes" that are difficult to interpret or explain

D. Calvaresi et al. (Eds.): EXTRAAMAS 2023, LNAI 14127, pp. 38–58, 2023.
https://doi.org/10.1007/978-3-031-40878-6_3

[13]. This lack of transparency and interpretability can create significant challenges, particularly in applications such as healthcare, finance, and criminal justice, where decisions can have profound consequences for human lives [4].

The current focus of XAI research is on developing techniques for "opening up" these black boxes and providing insights about how an intelligent system reached a particular decision or prediction [10]. This involves developing methods for visualising the internal workings of the system, such as feature importance scores, attention maps, or decision trees. In all such cases, the goal is to support the AI expert willing to figure out how the intelligent system works, rather than the non-expert user who wants to understand why the system is behaving in a particular way. Furthermore, and more importantly, all such methods are based on the assumption that software tools should aid humans' interpretation of the system. However, the expectations of the XAI community go beyond merely opening up black boxes. Ideally, XAI systems should be able to *automatically* provide explanations that surpass mere descriptions of how a system works [7]. Instead, they should offer insights into why the system is – or is not – behaving in a particular way, possibly, by autonomously interacting with the explainee.

To achieve this goal, XAI researchers are increasingly focusing on the automation and interactivity of the explanation process [8]. This involves developing AI systems that can generate explanations on the fly and adapt their explanations to the needs and knowledge level of the explainee [5]. Along this line, multi-agent systems (MAS) are likely the most adequate metaphor for intelligent explainable systems. There, interaction and autonomy are first-class citizens. Hence, explanation can be smoothly modelled as a multi-agent interaction, where the explainee and the explainer agent (either a human or a software agent [18]) interact to achieve a common goal, namely, providing a clear and effective explanation.

Accordingly, in this paper we focus on the general problem enabling the interaction between explainee and explainer agents. To address this problem, this paper proposes a general-purpose protocol for multi-agent based recommendation and explanations. The protocol specifies the roles and responsibilities of the explainee and the explainer agent and the types of information that should be exchanged between them to ensure a clear and effective explanation. Notably, our protocol builds on top of prior attempts to model explanations as multi-agent interactions, such as the work by [3]. The key features of our proposal are the *(i)* the separation of recommendations from explanations, and *(ii)* the support for contrastive explanations.

As a side contribution, the paper also describes the design of a Spade-based Python library implementing the proposed protocol—namely, PyXMas. It supports the plugging of different sorts of explanation strategies, and representations. This library can be used as a starting point for building intelligent explainable systems where both recommendation and explanation behaviours are delegated to individual agents. Overall, this paper represents an important step towards developing XAI systems that can provide automatic and interactive explanations.

2 Background and Related Works

This section briefly overviews the literature on recommender systems, with an emphasis on food recommender system and interactive/explainable recommendations.

2.1 Interactive Recommendation Systems

In the past, *interactive* recommender systems have received significant attention from the recommender systems researchers due to their ability to provide *personalised* recommendations to users dynamically based on their feedback and interactions [12]. The key point behind interactivity is getting feedback from the user during the recommendation session for the next recommendation. The authors of [6], for instance, follow a one-shot recommendation where a few questions learned offline from past observations (i.e., from previous sessions) are asked prior to the recommendation. Answers to these questions let the recommender system personalise and improve future recommendations.

Building interactivity, researchers has also started to incorporate explainability into recommender systems [3,21] in order to increase the transparency of a recommender system. They do so through a repeated recommendation session where the system also provides explanations (other than recommendations), and it gets feedback given the positive effect observed in more transparent recommendations [17]. For instance, in [15], the authors implement visual explanations to users for music recommendation using grouped bar charts in a live comparison of the user's specified preferences for six categories and the song's matching percentage of that category. Similarly, the authors of [20] propose a recommendation system mimicking a human salesman: the system applies conversational explanations to convince users to buy more fitting alternatives.

2.2 Prior Work on Explanation Protocols

This paper proposes an extension of the protocol introduced by [3], which is tailored on food recommendations and explanations. There, the user starts the interaction by providing their constraints, which include ingredients that they are allergic to (such as milk or peanuts), preferred or disliked ingredients (such as certain meats or vegetables), and the desired cuisine type (like Middle Eastern or Mexican). The agent responds by suggesting a recipe and providing an explanation. The user then can accept the recommendation, decline it, or provide feedback on either the recipe, the explanation, or both at once. Accordingly, the agent may generate a new recipe or provide further explanation, or both at once. This interaction continues in a turn-taking fashion until the user accepts, leaves the session early, or reaches a time limit. Figure 1 illustrates how agents interact in line with this protocol.

Summarising, the key contribution of [3] is a framework for creating a nutrition-related personalised recommender system that simultaneously produces recommendations and explanations. On the one hand, presenting recommendations and explanations altogether establishes a transparent interaction

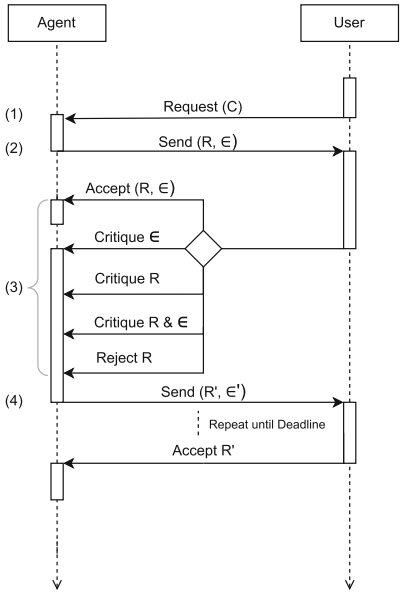


Fig. 1. FIPA representation of the negotiation protocol presented in [3]

with the user and may lead the user to accept such recommendations. On the other hand, unrequested explanations could be perceived as redundant and create an additional cognitive load on the user. The latter point has been studied, for instance, by Mualla *et al.* in [16]. There, *parsimony* has been outlined as one of the key features allowing successful human-agent interaction. In particular, parsimonious explanation are defined as the least complex that describes the situation adequately.

Accordingly, this work aims to revise the design to generate explanations based on users' requests, letting the user decide *when* to receive an explanation rather than providing a one-for-all solution. Furthermore, our revised protocol supports "zooming" explanations, where further explanatory details are only presented if and when the explainee is asking for them. In this way, the protocol lets users dynamically decide what degree of parsimony is fine for them

2.3 SPADE: Multi-agent Programming in Python

SPADE[1] is an open-source multi-agent system platform developed in Python. It provides a programming library for developing and simulating intelligent agents in various environments. The library is designed to be highly modular and extensible, allowing developers to easily create agents that can interact with each other and their environment.

[1] https://spade-mas.readthedocs.io.

At the modelling level, SPADE design and architecture are very close to the ones of JADE [2]. Accordingly, SPADE systems are distributed systems composed by agents which may or may not lay on the same network node. Agents' activities are governed by a set of behaviours, which are executed concurrently by the agent. Both agents and behaviours are implemented as abstract Python classes, which developers may extend to create their custom agents and behaviours.

Notable differences among SPADE and JADE mostly lay at the technological level. While JADE is a Java-based platform, SPADE is implemented in Python. This allows SPADE to be easily integrated with other Python libraries, there including the many ML and AI framework which are nowadays available for the Python platform. Furthermore, SPADE assumes agent interactions are mediated by an XMPP service, which is a standard protocol for instant messaging. This makes SPADE's agent communication facilities quite robust, interoperable, and scalable—as opposed to other agent platforms relying on proprietary or ad-hoc protocols. It also makes it easier to realise blended applications where agents interact with humans, other than with software agents.

Along this line, the SPADE framework comes with a range of features for developing intelligent agents, including communication protocols, message passing, and event handling. Notably, it supports the implementation of interaction protocols via finite-state machine behaviours—similarly to what JADE does.

Overall, the Spade library provides a powerful and flexible platform for developing intelligent agents, making it a popular choice for researchers and developers working in the field of multi-agent systems.

3 Explanation-Based Recommendation Protocol

The word "explanation" derives from the Latin word "explicare", which means "to unfold". There, the idea is that an explanation is a process of unfolding the meaning of a concept. Such a process is typically interactive, as it involves the interaction between some explainee and some explainer. In this sense, explanations is an inherently social protocol.

As far as interactions among human beings are concerned, the protocol is typically informal and unstructured. However, it typically involves an *explainee*, asking for help in understanding a given matter to some – allegedly, more knowledgeable – *explainer*. Explainers will then try their best to provide clear and effective explanations, possibly by trial-and-error, exploiting different explanation strategies or levels of detail. As interaction proceeds, explainers would also try to adapt their explanations to the needs and knowledge level of the explainee. Furthermore, explanations are commonly provided upon request, possibly in response to some prior information provided by either the explainee or the explainer, or someone else.

Modern intelligent systems are supposed to support decision-making by providing *recommendations*—possibly relying on artificial intelligence. Hence, when it comes to *explainable* intelligent systems, users commonly play the role of explainee, whereas software systems play both the role of the recommender and

the explainer. By adopting a multi-agent perspective, we can model both recommendation and explanation as a single interaction protocol among two agents—where one of the two (commonly, the explainee) is a human being [8]. Hence, we may interchangeably use the terms "explainee" and "user" (resp. "explainer" and "agent"). Accordingly, in this section, we propose a general-purpose interaction protocol for multi-agent based recommendation and explanation.

Our 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 on 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. Furthermore, explanations may be of two broad types, namely:

Ordinary explanations, which aim at answering the question "why did you recommend me this?";

Contrastive explanations, which aim at answering the question "why did you not recommend me that instead?".

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.

3.1 Abstract Formulation of the Protocol

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 exchanges among explainers and explanees, what information they should carry, and in which order they should be exchanged. Accordingly, the protocols relies on 13 types of messages, which may carry data fields of 5 different types, to be exchanged among agents playing 2 possible roles.

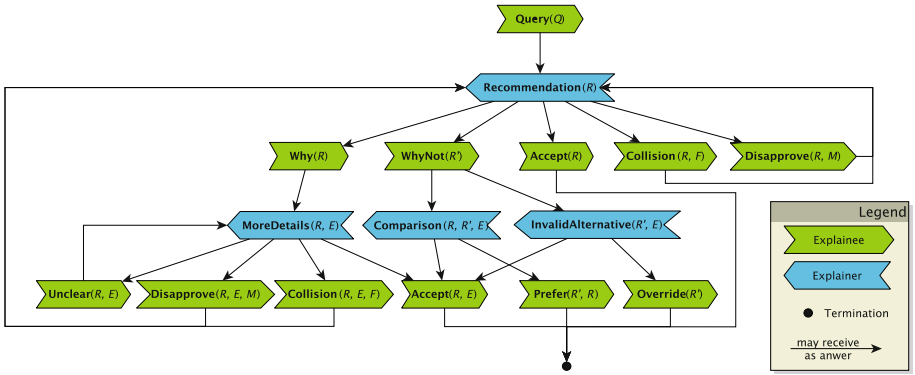


Fig. 2. Message communication diagram between an explainer agent (blue boxes) and an explainee (green boxes). Each box represents a message. Each message is connected to the ones it can receive as reply. (Color figure online)

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 which 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 specification for these types, tailoring them on 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 memorise 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: $\mathsf{Name}(Payload)$, 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*. A summary of message types and their admissible payloads in Fig. 2. Accordingly, message types are (description follows a breadth-first traversal the diagram in the figure):

1. $\mathsf{Query}(Q)$ is the message issued by the explainee to initiate the protocol: it carries a recommendation request Q;
2. $\mathsf{Recommendation}(Q, R)$ 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;
3. $\mathsf{Why}(Q, R)$ is the message issued by the explainee to request an explanation of a recommendation: it carries the original query Q and the recommendation R;
4. $\mathsf{WhyNot}(Q, R, R')$ 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;
5. $\mathsf{Accept}(Q, R, E?)$ 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;
6. $\mathsf{Collision}(Q, R, F, E?)$ 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;
7. $\mathsf{Disapprove}(Q, R, M, E?)$ 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;
8. $\mathsf{Details}(Q, R, E)$ 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;
9. $\mathsf{Comparison}(Q, R, R', E)$ 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;
10. $\mathsf{Invalid}(Q, R', E)$ 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;
11. $\mathsf{Unclear}(Q, R, E)$ 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;
12. $\mathsf{Prefer}(Q, R, R')$ 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;

13. $\mathsf{Override}(Q, R, R')$ 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.

Notably, messages are designed by keeping the representational sate transfer (ReST, [9]) architectural style into account. Hence, each message type is designed to carry all the information necessary for any involved party to decide which action to take next. This is the reason why all/most messages carry the original query Q and the recommendation R (or R') which they are referring to.

The message communication diagram from Fig. 2 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 summarise in the following. The explanation-based recommendation protocol consists in the following phases (depth-first traversal of Fig. 2):

1. the explainee initiates the protocol, by issuing a message $\mathsf{Query}(Q)$;
2. the explainer provides a message $\mathsf{Recommendation}(Q, R)$ in return;
3. the explainee may now:
   - 3.1 accept the recommendation, by answering $\mathsf{Accept}(Q, R)$, hence terminating the protocol;
   - 3.2 reject
     the recommendation because of M, by answering $\mathsf{Disapprove}(Q, R, M)$; or signal it as colliding with F, by answering $\mathsf{Collision}(Q, R, F)$. In this case, the explainer should propose another recommendation (go to 2.);
   - 3.3 ask for *ordinary* explanations, by answering $\mathsf{Why}(Q, R)$. In this case, the explainer should propose an explanation, by answering $\mathsf{Details}(R, E)$. The explainee may now:
     - 3.3.1. accept, reject, or signal R in light of E, by answering $\mathsf{Accept}(Q, R, E)$, $\mathsf{Disapprove}(Q, R, M, E)$, or $\mathsf{Collision}(Q, R, F, E)$, respectively, with outcomes similar to cases 3.1. and 3.2;
     - 3.3.2. ask for a better explanation via $\mathsf{Unclear}(Q, R, E)$ (go to 3.3.).
   - 3.4 ask for *contrastive* explanations motivating why not R', by answering $\mathsf{WhyNot}(Q, R, R')$. The explainer may now:
     - 3.4.1. explain the difference E among R and R', if R' is admissible w.r.t. its current knowledge base, by answering $\mathsf{Comparison}(Q, R, R', E)$. Now, the explainee may either (in both cases, the protocol terminates):
       - 3.4.1.1. accept R, via $\mathsf{Accept}(Q, R, E)$, or
       - 3.4.1.2. state that they prefer R', via $\mathsf{Prefer}(Q, R, R')$.
     - 3.4.2 explain that R' is not an admissible recommendation because of E, by answering $\mathsf{Invalid}(()Q, R', E)$. At this point, the explainee may either (in both cases, the protocol terminates):
       - 3.4.2.1. accept R, via $\mathsf{Accept}(Q, R, E)$, or
       - 3.4.2.2. override the explainer's decision, by stating that they prefer R', via $\mathsf{Override}(Q, R, R')$—hence forcing the explainer to update their own knowledge base accordingly.

3.2 Relevant Scenarios and Protocol Analysis

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 summarised in Fig. 3, and discussed below.

Quick Accept. This is the scenario depicted in Fig. 3a. There is no need for explanations, and the user simply accepts the recommendation provided by the agent.

For instance, the user asks for a restaurant recommendation, and the agent proposes a restaurant, and the user is fine with it.

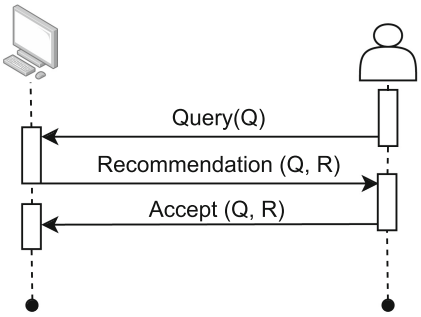
Quick Retry. This is the scenario depicted in Fig. 3b. There is no need for explanations, and the user simply rejects the recommendation provided by the agent, by either disapproving it or stating that it is in conflict with their own preferences. In both cases, the agent shall produce a new recommendation.

For instance, the user asks for a restaurant recommendation, and the agent proposes a steakhouse, but the user does not like it because: *(i)* they are vegetarian or they do not like steak, or *(ii)* they do not want to eat meat that day. In the former case, the user shall signal a collision among the recommendation and its preference—which the agent is expected to learn and to take into account for future recommendations. In the latter case, the user shall simply disapprove the recommendation—but the agent is not supposed to memorise such an event.

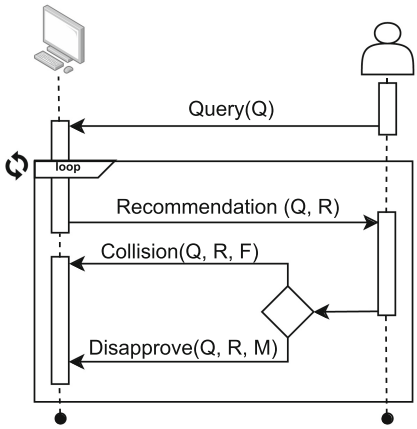
Ordinary Explanation Loop. This is the scenario depicted in Fig. 3c. The user asks for a recommendation, and the agent provides one. The user is not satisfied with the recommendation, and asks for an explanation. The agent provides an explanation, and the user is not satisfied with it. The user asks for further details, and the agent provides them. The loop may be repeated several times. Eventually, the user accepts the recommendation, or asks for a new one, similarly to the quick accept/retry scenarios.

The interesting part here is the explanation loop. It is a flexible mechanism, supporting zooming in/out explanations: the agent change the granularity of the explanations, by providing more or less details. For instance, the agent may provide *local* explanations first – i.e., explanations describing how the recommendation was produced – and then *global* explanations—i.e., explanations describing how recommendations are computed in general. The agent may also change the representation means of the explanations, by providing textual explanations first, and then visual explanations—or vice versa.

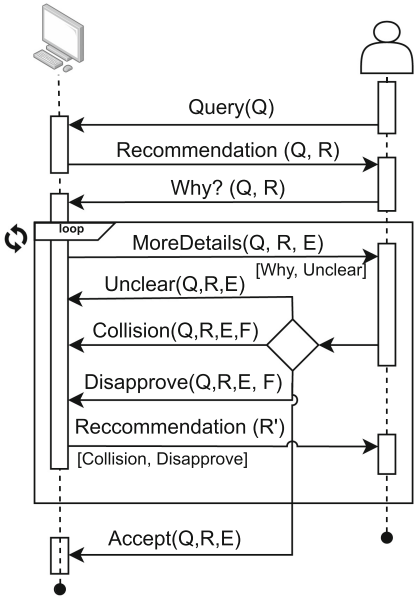
As an example, consider the situation where the user asks for a restaurant recommendation. The agent recommends some Asian restaurant in the users'



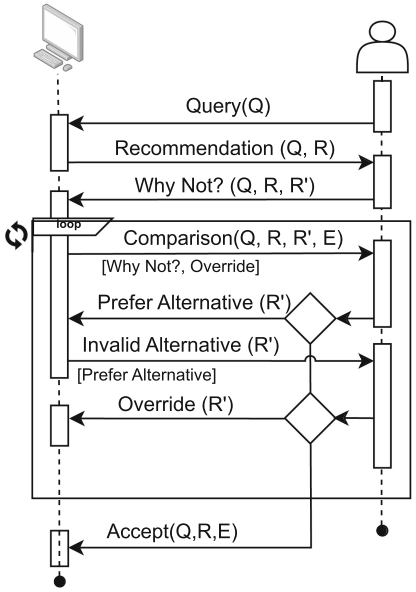
(a) Quick accept: the user accepts the recommendation without asking for explanations.



(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.

Fig. 3. Sequence diagrams describing most common scenarios of the protocol.

surroundings, having 4.3 stars (out of 5) on ACME-Advisor.com. The user is curious in understanding the reason why the agent proposed that restaurant, and asks for an explanation. The agent provides an explanation, stating that the restaurant is close to the user, and that it has a good rating. Furthermore, the agent reminds to the user that – to the best of its knowledge – they like Sushi. The user is still not satisfied, and asks for further details. The agent provides a textual explanation, stating that it commonly recommends the highest ranked restaurant matching the user's tastes, and having a distance which is not higher than 1 km. Eventually, the user may be satisfied with the explanation, and accept the recommendation; or they may reject the recommendation and possibly request a new one.

Contrastive Explanation Loop. This is the scenario depicted in Fig. 3d. The user asks for a recommendation, and the agent provides one (R). The user was not expecting that recommendation, but rather another one (R'), and they ask for a *contrastive* explanation. If the users' recommendation is acceptable as well, the agent provides a comparison between the two recommendations, arguing one of the two is better than the other. Otherwise, if the users' recommendation is not acceptable, the agent provides an explanation for why it is not acceptable. In both cases, the user may either accept the original recommendation or prefer their own—possibly overriding the agent's recommendation. In the case of an override, the agent should learn from the user's preferences, and possibly update its recommendation policy. In any case, the interaction ends.

The key points here are the possibility, for the user, to *(i)* simulate alternative recommendations, and *(ii)* contradict the recommender agent in order to let it learn.

Consider for instance the aforementioned restaurant recommendation case. The user may not be satisfied with the agent's recommendation concerning an Asian restaurant, and propose the local steakhouse instead. The agent may then either consider the proposal acceptable or not, depending on the dietary goals and physiological condition of the user. If the agent considers the proposal acceptable, it may provide a comparison between the two recommendations, stating that the Asian restaurant is closer. At this point, the user may either accept the original recommendation (Asian restaurant) or prefer their own (steakhouse). Otherwise, if the agent considers the proposal unacceptable, it may provide an explanation for why it is not acceptable—e.g. steak is violating the user's dietary goals. In this case, the user may either accept the original recommendation (Asian restaurant) or override it (steakhouse).

3.3 Which Sorts of Explanations and Recommendation?

The explanation protocol is agnostic w.r.t. the particular way in which explanations and recommendations are represented. Indeed, it is implementers' responsibility to define the representation means of explanations and recommendations—other than deciding how they should be computed in practice. The protocol simply dictates *when* explanations and recommendations should be computed.

Accordingly, in this subsection we provide a few insights about the possible design choices for explanations and recommendations.

Recommendations are commonly supported by means of one or more ML predictors, trained on users' data. Whether predictors' training is a responsibility of the recommender agent, or simply the agent is endowed with pre-trained predictors at deployment time, is an implementation detail. In either cases, the recommender agent is supposed to know (or be able to access or acquire) profile-related information about the user. Such information may come, for instance, from some initial configuration phase, as well as be inferred by the agent itself, from the accepted/rejected recommendations. To support the latter case, the agent should be endowed with some learning algorithm, making it able to (re)train the predictors when new user data is avaliable. Under this perspective, as far as recommendations are concerned, the explainer agent is simply proxying the ML predictor(s).

Explanations, on the other hand, are not necessarily supported by ML predictors. In this case, the XAI literature is full of possible approaches, including both visual, textual, and numeric explanations. The interested reader may refer to high impact surveys such as [1,10] for a comprehensive overview of the state of the art. The key point here is that the explainer agent should not only wrap the ML predictor(s), but also encapsulate the logic for representing and computing explanations.

Along this line, one critical situation is the one where recommendations and explanations come with different representation means—e.g. textual and visual. In this case, the explainer agent should be able to bridge the gap between the two, by providing a unified representation of the recommendation and its explanation.

To mitigate this issue, designers may consider adopting computational logic as the reference framework for both recommendations and explanations. In computational logic, both knowledge bases, and queries, are represented as logic formulæ. Logic formulæ, in turn, can be exploited to represent both recommendations and explanations. In fact, logic queries may be used to represent recommendation requests, and logic solutions may be used to represent recommendations, whereas proof trees may be exploited to compute explanations.

For instance, a recommendation query may consist of the logic goal $\mathit{should\_eat}$ $(\mathit{Food}, \texttt{lunch})$, where Food is a logic variable, i.e., a placeholder for unknown values. Recommendations $R, R', R'', \ldots$ may be logic solutions, i.e., assignments of logic variables (e.g., $\mathit{Food} = \texttt{paella}$). Explanations $E, E', E'', \ldots$ may be of many sorts:

- local explanation, e.g. the path in the proof tree computed by the explainer agent to provide the recommendation;
- global explanation, e.g. the logic program used by the explainer agent to provide the recommendation;
- contrastive explanation, e.g. quality metrics comparing two or more recommendations, or the violated constraints making some recommendation unacceptable;
- any combination of the above.

User features $F, F', F'', \ldots$ may be raw facts describing the user (e.g., *age*(`31`), *goal*(`lose_weight`), *category*(`vegetarian`)). Disapprove motivations may be predefined facts such as `dislike` – the user does not like a recommendation and the agent should learn that – or `not_now`—the user simply does not want that recommendation now, may they like it in general (so the agent should not memorise that).

4 From Theory to Practice with PyXMas

In this section, we describe how our protocol can be reified into actually usable agent-oriented software. Accordingly, we discuss the design of PyXMas[2], i.e., our Python library for explainable multi-agent systems.

PyXMas is an agent-oriented software library based on Spade. It comes with predefined – yet parametric – implementation of the protocol described in Sect. 3, in the form of reusable agent behaviours. In this way, researchers and developers can easily take advantage of our protocol to build explainable MAS, without wasting time in re-implementing the protocol. Rather, they can focus on the design of the actual recommender and explainer agents, as well as on the representation means of recommendations and explanations. In particular, PyXMas requires designers to define which particular notion of recommendation and explanation they want their agents to support, and how should agents compute or react to them.

4.1 PyXMas Architecture

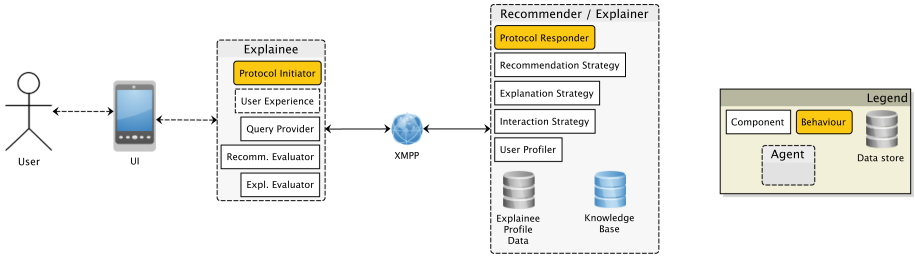


Fig. 4. Architecture of PyXMas. Behaviours are provided by the library, and they are parametric w.r.t. *components*. PyXMas users are responsible for implementing and plugging their own components, in order to tailor PyXMas to their specific needs. Optionally, if the explainee agent is human, users may also need to implement a UX component supporting interaction with the explainee—possibly via some device.

Figure 4 summarises the modular architecture of PyXMas. Leveraging on Spade facilities, PyXMas is implemented two behaviours, which can be easily

[2] https://github.com/pikalab-unibo/pyxmas.

plugged into any agent—namely, the protocol *initiator* and the *responder*. On the one hand, the initiator behaviour is responsible for sending recommendation queries to the explainer agent, and for receiving and processing the corresponding recommendations and explanations. Consequently, the initiator behaviour is meant to be plugged into the explainee agent. On the other hand, the responder behaviour is responsible for receiving requests from the initiator, and for computing and sending recommendations and explanations. Consequently, the responder behaviour is meant to be plugged into the explainer agent.

Figure 4 also shows that PyXMas is designed to be highly parametric. In particular, the initiator and responder behaviours are parametric with respect to a number of *components* which dictate the actual behaviour of the agents. In this way, users of PyXMas can easily tailor the library to their specific needs, by implementing and plugging their own components.

Explainer Agent. As far as the explainer agent is concerned, the responder behaviour requires the following components provided by developers:

Recommendation Strategy— the component in charge of computing recommendations for any given query. In addition to users' requests and feedback, the recommendation strategy should consider profile information about users (e.g., their goals/interests, such 'losing weight'), as well as their preferences and interests (e.g., vegetarian users do not eat meat). Agents may have limited information about their users' preferences but could learn more over-time—also thanks to our protocol. The learned preferences and constraints could be exploited to generate well-targeted recommendations.

Explanation Strategy—the component in charge of computing explanations for any given recommendation. This is where designers can develop different approaches for generating explanations supporting the given recommendations. While operating, the explanation strategy may exploit the estimated user profiles (e.g., the user dislikes animal-derived food), as well as commonsense or background information (e.g., food X only contains vegetable-derived ingredients) which is made available to the agent.

User Profiler—the component in charge of learning user profiles from users' feedback. This component may adopt any heuristic-based or machine learning approach to learn users' preferences over time. As the explainer agent interacts with the explainee, it gets (possibly implicit) feedback about the recommendations it provides. It this way, it may infer valuable information about their preferences and interests. In this way, the agent can update the explainee's profile information—in order to eventually provide better explanations or recommendations.

Interaction Strategy—the component in charge of which recommendation and explanation strategies to exploit, and how to present recommendations and explanations to the explainee. This component is responsible for processing the content of the exchanged messages and transferring them to the related components. Note that not only the content of the messages but also how these messages are expressed/represented plays a crucial role in interactive

intelligent systems. Consider for instance the case where the explainer agent is a humanoid robot. In the case, the interaction component is in charge of selecting the best gestures or facial expressions supporting the action taken (e.g., a surprising facial expression when it discovers unexpected knowledge about the user). The interaction strategy component may also operate the other way around. For instance, if the robot can sense people facial emotion, tones, or gestures, it may adjust other components behaviour accordingly.

It is worth mentioning that, to operate correctly, the explainer agent is supposed to collect and store two sorts of information, namely: *(i)* profile data about the explainee, and *(ii)* common-sense/background knowledge about the domain of interest. For an architectural perspective (cf. Fig. 4), information of these sorts are store in to *ad-hoc* data stores. In particular, profile data is stored in the *user profile* data store, while the common-sense/background knowledge is stored in the *knowledge base* data store. These data stores are local w.r.t. the explainer agent, and act as its memory/belief base. They are subject to reads/updates by the different components of the explainer agent.

Explainee Agent. As far as the explainee agent is concerned, the initiator behaviour requires the following components provided by developers:

Query Provider—the component in charge of generating queries for the explainer agent, depending on the current goals of the explainee.

Recommendation Evaluator—the component in charge of evaluating the recommendations provided by the explainer agent and deciding whether to accept or reject them.

Explanation Evaluator—the component in charge of evaluating the explanations provided by the explainer agent, and affecting the recommendation evaluator accordingly.

In the particular case where the explainee agent is a human user, the explainer agent should be implemented as a simple proxy agent, which acts on behalf of the human and mediates their interaction with the recommender agent. In that case, the proxy agent is responsible for human-computer interaction, possibly via some user interface (UI) presented to the human on top of some device (e.g., a smartphone). This is the situation depicted in Fig. 4. When this is the case, the proxy agent is supposed to include one further component, namely the **User Experience** (UX) one.

When present, the UX component is in charge of governing the UI, hence grasping humans' inputs and presenting recommendations and explanations to them. In this case, the other components are simply in charge of processing the humans' inputs and generating the appropriate messages to be sent to the explainer agent.

4.2 PyXMas Design

PyXMas consists of a Python library providing:

- abstract classes defining the (de)serialisation message payloads exchanged between the explainer and explainee agents (cf. Sect. 3.1),
- abstract classes defining the initiator and responder behaviours.

In both cases, we exploit abstract classes as we leave room for costumisation. In fact, developers may want to extend the provided abstract classes and override specific methods to plug their own components.

Accordingly, in this subsection, we describe the abstract classes available in PyXMas and the way they are supposed to be extended to so ars to build some actual explainable MAS.

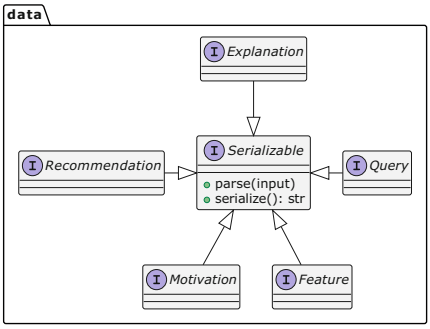
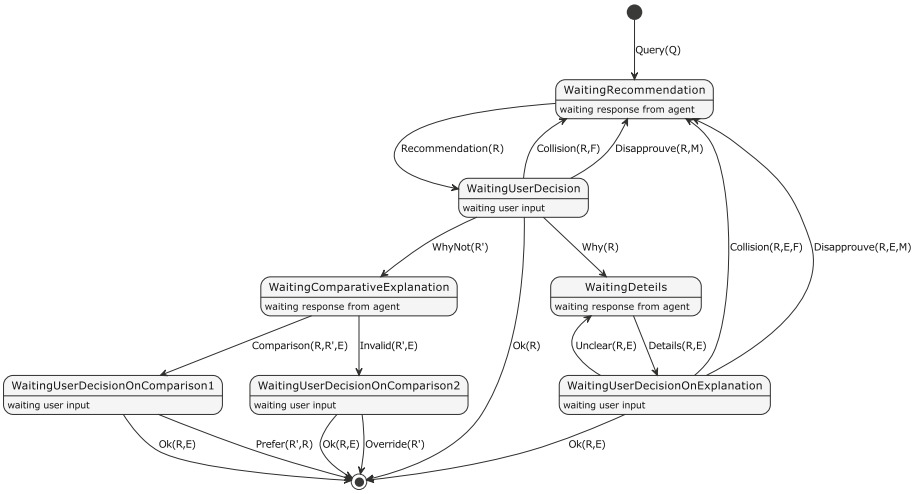


Fig. 5. Abstract classes for message payloads in PyXMas.

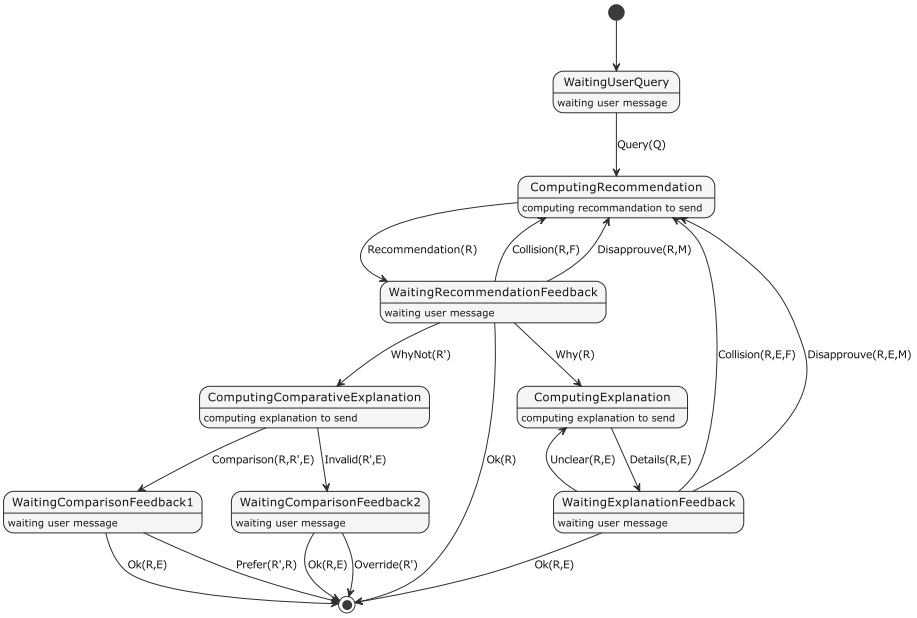
Data Types for Message Payloads. As shown in Fig. 5, PyXMas provides 5 abstract classes for the as-many data types defined in Sect. 3.1. These classes simply force developers to make these data types *serialisable*—i.e., to support their conversion into/from strings. This is necessary for the explainer and explainee agents to exchange messages over the network.

How to actually represent queries, recommendations, explanations, and so on, is left to developers. In fact, the only constraint is that the serialised version of these data types should be both machine- and human-interpretable.

When it comes to design some actual explainable MAS, developers may plug their custom notion of query, recommendation, explanation, and so on, by extending the provided abstract classes, and by implementing their (de)serialization-related methods.



(a) Initiator-side state diagram.



(b) Responder-side state diagram.

Fig. 6. State diagrams describing the initiator and responder behaviours as implemented in PyXMas.

Predefined Behaviours. PyXMas also provides two abstract classes for as many protocol roles agents may play, namely: the initiator and the responder behaviours. Building on Spade facilities, these classes are technically finite-state machine (FSM) behaviours. In other words, they are implemented as a set of states, each of which is associated with a set of actions to be performed when the agent enters that state. Figure 6 shows the state diagrams describing the initiator and responder behaviours as implemented in PyXMas. Broadly speaking, states either represent situations where agents are waiting for messeges from the other side, or situations where one of the agent is busy computing (resp. evalutation) a message for (resp. from) the other side.

These classes come with template methods (a.k.a. callbacks) that developers may override to plug their own components. In particular, on the initiator side, callbacks are supposed to be overridden to control how the explainee agent: *(i)* generates queries, *(ii)* evaluates recommendations and decides whether to accept or reject them, *(iii)* evaluates explanations, and decides whether to accept or reject recommendations accordingly. On the responder side, callbacks are supposed to be overridden to control how the explainer agent: *(i)* generates recommendations, *(ii)* generates explanations, *(iii)* handles situations where recommendations are accepted/rejected.

5 Conclusion and Future Work

In this paper, we present a general-purpose protocol for explainable MAS. The protocol is based on the idea that explanations should be provided upon request, by letting the same intelligent agent that is responsible for the recommendation process explain its own decisions. This subtends the existence of another entity – namely, the *explainee* agent – which requests recommendations and, possibly, explanations to the aforementioned intelligent agent. Under such hypotheses, our protocol regulates the interaction among such agents.

Despite our formulation is abstract, we discuss how concrete sorts of recommendations and explanations could be modelled and exchanged via the proposed protocol. Along this line, we also provide a Python implementation of the protocol – namely, PyXMas –, which is available as an open-source library on GitHub. PyXMas supports the pluggability of custom recommendation/explanation definitions—hence making it possible to re-use the protocol in different contexts.

Future works. Our protocol, as well as the PyXMas technology, plays a crucial role in the context of the Expectation project [5]—which is funding this work. There, the exploitation of multi-agent interaction as a means for explaining recommendations is at the core of the project.

Accordingly, further research is needed to investigate how the protocol impacts human-user interaction as a means for XAI. Along this way, we are planning both theoretical extensions of the protocol and technical improvements of the PyXMas technology—possibly enabling empirical studies on the impact on explainability.

In particular, concerning the protocol, we are planning to extend the formulation to support meta-data describing the emotional state of the explainee agent—hence studying of such meta-information may affect the recommendation/explanation process.

Concerning the PyXMas technology, we are planning to support the exploitation of symbolic knowledge extraction [19] and injection [14] as a means for explaining recommendations. This would imply leveraging on symbolic AI techniques to represent explanations and recommendations. Finally, we plan to provide better support towards human-computer interaction based on PyXMas. In this regard, our intention is to develop a Web- or Telegram-based graphical user interface for letting humans interact with PyXMas agents.

Acknowledgements. This work has been supported by the Chist-Era IV project "Expectation", the Italian Ministry for Universities and Research (G.A. `CHIST-ERA-19-XAI-005`), and by the Scientific and Research Council of Turkey (TÜBİTAK, G.A. `120N680`).

References

1. Barredo Arrieta, A., et al.: Explainable explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf. Fusion **58**, 82–115 (2020). https://doi.org/10.1016/j.inffus.2019.12.012
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Hoboken (2007). http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470057475.html
3. Buzcu, B., Varadhajaran, V., Tchappi, I., Najjar, A., Calvaresi, D., Aydogan, R.: Explanation-based negotiation protocol for nutrition virtual coaching. In: Aydogan, R., Criado, N., Lang, J., Sánchez-Anguix, V., Serramia, M. (eds.) PRIMA 2022: Principles and Practice of Multi-Agent Systems - 24th International Conference, Valencia, Spain, 16–18 November 2022, Proceedings. Lecture Notes in Computer Science, vol. 13753, pp. 20–36. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-21203-1_2
4. Calegari, R., Ciatto, G., Omicini, A.: On the integration of symbolic and sub-symbolic techniques for XAI: a survey. Intelligenza Artificiale **14**(1), 7–32 (2020). https://doi.org/10.3233/IA-190036
5. Calvaresi, D., et al.: Expectation: personalized explainable artificial intelligence for decentralized agents with heterogeneous knowledge. In: Calvaresi, D., Najjar, A., Winikoff, M., Främling, K. (eds.) EXTRAAMAS 2021. LNCS (LNAI), vol. 12688, pp. 331–343. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-82017-6_20
6. Christakopoulou, K., Radlinski, F., Hofmann, K.: Towards conversational recommender systems. In: KDD 2016: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 815–824 (2016). https://doi.org/10.1145/2939672.2939746
7. Ciatto, G., Calegari, R., Omicini, A., Calvaresi, D.: Towards XMAS: eXplainability through multi-agent systems. In: Savaglio, C., Fortino, G., Ciatto, G., Omicini, A. (eds.) AI&IoT 2019 - Artificial Intelligence and Internet of Things 2019, CEUR Workshop Proceedings, vol. 2502, pp. 40–53. Sun SITE Central Europe, RWTH Aachen University (2019). http://ceur-ws.org/Vol-2502/paper3.pdf

8. Ciatto, G., Schumacher, M.I., Omicini, A., Calvaresi, D.: Agent-based explanations in AI: towards an abstract framework. In: Calvaresi, D., Najjar, A., Winikoff, M., Främling, K. (eds.) EXTRAAMAS 2020. LNCS (LNAI), vol. 12175, pp. 3–20. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51924-7_1
9. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. ACM Trans. Internet Technol. **2**(2), 115–150 (2002). https://doi.org/10.1145/514183.514185
10. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. **51**(5), 93:1–93:42 (2018). https://doi.org/10.1145/3236009
11. Gunning, D.: Explainable artificial intelligence (XAI). Funding Program DARPA-BAA-16-53, DARPA (2016). http://www.darpa.mil/program/explainable-artificial-intelligence
12. Knijnenburg, B.P., Willemsen, M.C., Hirtbach, S.: Receiving recommendations and providing feedback: the user-experience of a recommender system. In: Buccafurri, F., Semeraro, G. (eds.) EC-Web 2010. LNBIP, vol. 61, pp. 207–216. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15208-5_19
13. Lipton, Z.C.: The mythos of model interpretability. Commun. ACM **61**(10), 36–43 (2018). https://doi.org/10.1145/3233231
14. Magnini, M., Ciatto, G., Omicini, A.: On the design of PSyKI: a platform for symbolic knowledge injection into sub-symbolic predictors. In: Calvaresi, D., Najjar, A., Winikoff, M., Främling, K. (eds.) Explainable and Transparent AI and Multi-Agent Systems, 4th International Workshop, EXTRAAMAS 2022, Virtual Event, Revised Selected Papers, Lecture Notes in Computer Science, 9–10 May 2022, vol. 13283, chap. 6, pp. 90–108. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-15565-9_6
15. Millecamp, M., Htun, N.N., Conati, C., Verbert, K.: To explain or not to explain: The effects of personal characteristics when explaining music recommendations. In: IUI 2019: Proceedings of the 24th International Conference on Intelligent User Interfaces, pp. 397–407. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3301275.3302313
16. Mualla, Y., et al.: The quest of parsimonious XAI: a human-agent architecture for explanation formulation. Artif. Intell. **302**, 103573 (2022). https://doi.org/10.1016/j.artint.2021.103573
17. O'Donovan, J., Smyth, B., Gretarsson, B., Bostandjiev, S., Höllerer, T.: PeerChooser: visual interactive recommendation. In: CHI 2008: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1085–1088 (2008). https://doi.org/10.1145/1357054.1357222
18. Omicini, A.: Not just for humans: explanation for agent-to-agent communication. In: Vizzari, G., Palmonari, M., Orlandini, A. (eds.) AIxIA 2020 DP – AIxIA 2020 Discussion Papers Workshop. AI*IA Series, vol. 2776, pp. 1–11. Sun SITE Central Europe, RWTH Aachen University, Aachen (2020). http://ceur-ws.org/Vol-2776/paper-1.pdf
19. Sabbatini, F., Ciatto, G., Calegari, R., Omicini, A.: Symbolic knowledge extraction from opaque ML predictors in PSyKE: platform design & experiments. Intelligenza Artificiale **16**(1), 27–48 (2022). https://doi.org/10.3233/IA-210120
20. Shimazu, H.: ExpertClerk: a conversational case-based reasoning tool for developing salesclerk agents in e-commerce webshops. Artif. Intell. Rev. **18**, 223–244 (2002). https://doi.org/10.1023/A:1020757023711
21. Zhang, Y., Chen, X.: Explainable recommendation: a survey and new perspectives. Found. Trends Inf. Retr. **17**(1), 1–101 (2020). https://doi.org/10.1561/1500000066