Exploiting Answer Set Programming for Building Explainable Recommendations

Erich Teppan[1] and Markus Zanker[2]

[1] Universität Klagenfurt, Austria
[2] Free University of Bolzano, Italy

Abstract. Explainability, i.e. the capability of a recommendation system to justify its proposals, becomes an ever more important aspect in light of recent legislation and skeptic users. Answer Set Programming (ASP) is a logic programming paradigm aiming at expressing complex problems in a succinct manner. Originating in the area of deductive databases ASP has settled down as a strong alternative to constraint programming and other declarative approaches for solving hard combinatorial problems. Due to its rich set of high level language constructs such as weak constraints or aggregates it turns out that ASP is also perfectly suitable for realizing knowledge and/or utility-based recommendation applications. Hereby, every aspect of such a utility-based recommendation engine can be specified within ASP including the specification of user and product attributes, the calculation of utility values as well as requirements relaxation in case that no recommended product item fully matches the user needs and explanations. In this paper we give an introduction to the concepts of ASP and how they can be applied in the domain of recommender systems. Based on a small excerpt of a real life recommender database we exemplify how utility based recommendation engines can be implemented with just some few lines of code and show how meaningful explanations can be derived out of the box.

Keywords: recommender systems · explanations · answer set programming

1 Introduction

The ability of a recommendation system to explain to users why a specific item is recommended has reached considerable research momentum, supported also by recent legislation like GDPR [3] that even codifies a right for explanations of the outcomes of algorithmic decision makers. Although knowledge-based recommendation strategies [9] are a niche topic compared to predominant machine learning based approaches, they are nevertheless in actual use for high-involvement product domains like financial services [4] or consumer products where many variables and aspects are typically considered during decision making processes. Utility-based recommender systems can be considered to be a specific variant

[3] See https://eugdpr.org/ for reference.

of knowledge-based systems, where the matching between user preferences or needs and item properties or features is realized via utility functions. The recent revived attention for knowledge-based approaches [1] is actually based on three pillars: the wide availability of structured and unstructured content that can be exploited for knowledge extraction, the promise of achieving beyond accuracy goals [11] like transparency, validity and explainability as well as the development of ever more efficient computational mechanisms for processing declarative knowledge. The answer set programming (ASP) paradigm under the stable model semantics [7] must be seen in the context of the latter pillar. It is a successful logic programming (LP) paradigm that has evolved from an academic discipline rooted in deductive databases to an approach that is practically applicable in many different domains [6]. It turns out that ASP, due to its rich set of different language constructs, seems to be perfectly appropriate for also expressing recommendation problems, particularly those ones that can be naturally modeled based on a knowledge or utility-based recommendation paradigm. One big advantage of encoding a recommendation engine in ASP is the compact high level representation that eases maintenance. Another advantage is that explanations for recommendations can be derived quite naturally as a side product of recommendation calculation based on logic rules.

In this paper, we introduce for the first time how the ASP mechanism can be used to build the complete logic of a utility-based recommender system. Hereby, we focus particularly on how to automatically derive meaningful explanations for recommendations. As a first proof of concept we use an excerpt from a real world knowledge base from the financial service domain [3] and show how a corresponding recommendation engine, in particular calculations of item utilities and explanations, can be expressed in ASP in a succinct way.

2 Related Work

To the best of our knowledge there exist only two papers employing ASP in the context of recommender systems. In [10] the authors propose dynamic logic programming, an extension of ASP, in order to provide a user the means to specify changes of their user profile in order to support the recommendation engine in producing better recommendations. Clearly, the focus of this approach is totally different as it is not concerned with building any aspect of the recommendation engine itself.

Not so for the second paper identified. The system described in [8] consists of two ASP based components. The first component is used for automatic extraction of relevant information from touristic offers contained in leaflets produced by tour operators. This information is in turn added to a tourism ontology, i.e. the second component. The second component is responsible for answering the question which touristic offers match a certain customer profile that is also added to the ontology. Thus, the system described in [8] can be seen as a raw content-based recommender. In contrast to that, the approach discussed herein is different from several aspects. First, we do not build upon ontologies, but di-

rectly harness the ASP knowledge representation. Second, beyond making solely the binary choice, whether an item is matching the user preferences or not, we calculate fine grained utilities to rank items. Third, we demonstrate meaningful explanations following the taxonomy of [5], where explanations actually reason on all three information categories: user preferences, item properties as well as comparisons with alternatives.

3 Working Example

In order to illustrate the applicability of ASP for knowledge-based recommendation approaches we develop a motivating example. It builds on a utility-based recommendation scenario, where the domain knowledge is encoded by relating user needs and item properties via utility values on different dimensions based on multi-attribute utility theory (MAUT) [2]. Due to the clarity of the dependencies we build on a published example [3] from the domain of financial services that actually constitutes a small excerpt from a real world recommender system in use by a European financial services provider.

The core of MAUT specifications are utility dimensions. These dimensions are abstract concepts which contribute to the overall utility of an item as perceived by a user. In our working example there are two dimensions: *profit*, i.e. the financial return of an investment, and *availability*, i.e. how quickly an investment can be converted back to cash. Note, that these utility dimensions are abstract concepts capable to model and map also indirect relationships between user and/or product attributes. In our example, the simplified user profiles contain just two attributes (or explicit preferences), namely the *duration* of an investment and the personal *goal* of a user with respect to some investment. For the investment *duration* let us assume three distinct values *long*, *medium* and *short* term. Analogously, for the *goal* description we again assume three values: *savings* for a rainy day, profitable *growth* and *venture*. Similarly, properties also describe product items. Let us assume that financial products can be described based on the portions of *shares* a product contains (0%, $\leq 30\%$, $\leq 60\%$, $\leq 80\%$, $> 80\%$) as well as expected price *fluctuation* (*low*, *medium* and *high*).

The connection between utility dimensions and attribute values can be given by so called scoring values (or just scores for short) that specify how much user preferences and product properties contribute to the utility dimensions. The higher the score for a dimension based on a given attribute value the more this attribute contributes to fulfilling this dimension. In this example we use scores between 0 and 10 where a score of 0 signifies that an attribute value does not positively contribute to a dimension. Tables 1 and 2 depict the scores for the user attribute values with respect to these utility dimensions. Tables 3 and 4 depict the example scores for the product attribute values with respect to the utility dimensions. For demonstration purposes, we have a small set of four financial products. Table 5 depicts the attribute values of these products.

duration	**profit**	**availability**
long	10	2
medium	7	6
short	3	10

Table 1. User scores for *duration* dimension

goal	**profit**	**availability**
savings	2	8
growth	6	4
venture	10	2

Table 2. User scores for *goal* dimension

shares	**profit**	**availability**
0%	2	7
$1-30\%$	4	6
$31-60\%$	5	5
$61-80\%$	8	2
$81-100\%$	10	1

Table 3. Product scores for *duration* dimension

fluctuation	**profit**	**availability**
high	7	4
medium	5	6
low	1	8

Table 4. Product scores for *goal* dimension

product	**shares**	**fluctuation**
mutual fund AXP	31-60%	medium
bond BX	0%	medium
bond RX	0%	high
mutual fund CXP	61-80%	high

Table 5. Product attribute values

For a given product, its contribution to a dimension is calculated as the sum of its attribute value scores on that dimension. Table 6 depicts the computation for the example products.

product	**profit**	**availability**
mutual fund AXP	5+5=10	5+6=11
bond BX	2+5=7	7+6=13
bond RX	2+7=9	7+4=11
mutual fund CXP	8+7=15	2+4=6

Table 6. Product contributions to utility dimensions

Likewise, the weight that is given to a dimension is estimated as the sum of user attribute value scores on that dimension. For the example user profile depicted in Table 7 the score calculation is given in Table 8.

profile	**duration**	**goal**
Simon	medium	savings

Table 7. Example user profile

Finally, the overall utility of each product for a specific user is defined by the sum of weighted dimension contributions. The computations for our toy example are given in Table 9.

Hence, for this example scenario, the top ranked product for user *Simon* would be *bondBX* followed by *mutual fund AXP*. These overall utilities can be utilized already for a very basic justification for each recommendation, such as 'Item X promises the highest overall utility u_x'. However, this might not yet satisfy users, who would like to understand more what happens behind the scenes. A closer look to the dimensional contributions and weights provides us means for further argumentation, for instance, 'The top ranked *bond BX* scores also strongest in terms of *availability*, which is also considered most relevant given your preferences p_u'. On the other hand, *mutual fund AXP* scores second best on both dimensions, i.e. it can be justified as being a good compromise. Although *mutual fund CXP* is the worst overall it shows the best contributions to *profit*. For *bond RX* there is, for instance, a clear *negative* explanation why

profile	profit	availability
Simon	7+2=9	6+8=14

Table 8. User weights for utility dimensions

user	product	utility
Simon	mutual fund AXP	9*10+14*11=244
	bond BX	9*7+14*13=245
	bond RX	9*9+14*11=235
	mutual fund CXP	9*15+14*6=219

Table 9. Product utilities for the example user profile

it must not be recommended since it is dominated by *mutual fund AXP* that offers the same utility score in terms of *availability*, but even more in terms of *profit*. Thus, there is no rational reason for opting towards *bond RX*.

These are just some examples of explanations that can be generated based on utilities. In the next section we show how to express the whole utility and explanation calculation, i.e. the complete recommendation engine, in ASP.

4 ASP for recommenders

In this section we show how utility based recommendation and explanation can be expressed based on the answer set programming (ASP) paradigm in form of first order logic facts and rules. Since it is not possible at this point to give a complete introduction to ASP, we explain the ASP code snippets on an intuitive level[4].

Listing 1.1 depicts how the scoring and product data in Table 1-5 can be expressed by logic facts.

```
user_score(duration,long,profit,10).
user_score(duration,medium,profit,7).
user_score(duration,short,profit,3).
user_score(duration,long,availability,2).
user_score(duration,medium,availability,6).
user_score(duration,short,availability,10).

user_score(goal,savings,profit,2).
user_score(goal,growth,profit,6).
user_score(goal,venture,profit,10).
user_score(goal,savings,availability,8).
user_score(goal,growth,availability,4).
user_score(goal,venture,availability,2).

product_score(shares,"0",profit,2).
product_score(shares,"1-30",profit,4).
```

[4] For an in-depth introduction to ASP please refer to [6].

```
product_score(shares,"31-60",profit,5).
product_score(shares,"61-80",profit,8).
product_score(shares,"81-100",profit,10).
product_score(shares,"0",availability,7).
product_score(shares,"1-30",availability,6).
product_score(shares,"31-60",availability,5).
product_score(shares,"61-80",availability,2).
product_score(shares,"81-100",availability,1).

product_score(fluctuation,high,profit,7).
product_score(fluctuation,medium,profit,5).
product_score(fluctuation,low,profit,1).
product_score(fluctuation,high,availability,4).
product_score(fluctuation,medium,availability,6).
product_score(fluctuation,low,availability,8).

product("mutual fund AXP",shares,"31-60").
product("mutual fund AXP",fluctuation,medium).
product("bond BX",shares,"0").
product("bond BX",fluctuation,medium).
product("bond RX",shares,"0").
product("bond RX",fluctuation,high).
product("mutual fund CXP",shares,"61-80").
product("mutual fund CXP",fluctuation,high).
```

Listing 1.1. Scoring and product data given as logic facts

Analogously, Listing 1.2 shows how the user profile data from Table 7 is encoded by ASP logic.

```
profile("Simon",duration,medium).
profile("Simon",goal,savings).
```

Listing 1.2. User profile given as logic facts

In an interactive recommendation scenario, these *profile* facts are to be changed conforming to actual user requirements or query.

```
contribution(Productname,Dimension,Total):-
 product(Productname,_,_),
 product_score(_,_,Dimension,_),
 Total=#sum{Score,Attribute:
  product(Productname,Attribute,Value),
  product_score(Attribute,Value,Dimension,Score)}.
```

Listing 1.3. Calculation of product contributions in ASP

For calculating the contributions of all products for all utility dimensions only the single ASP rule given in Listing 1.3 is needed. Like also in other logic programming languages (e.g. Prolog) the ':-' operator stands for left implication. To put it very simple, the left hand side of the rule (i.e. left from ':-') specifies the atoms that are to be added to the solution, which in ASP is called answer set,

based on the calculations done on the right hand side. The core of the calculation is performed by a *#sum* aggregate that, sums up all scoring values for a product *Productname* on a dimension *Dimension*. Terms beginning with capital letters like *Productname* or *Dimension* stand for logic variables. Consequently, this rule 'fires' once for each combination of products and dimensions.

The atoms produced by the rule in Listing 1.3 and included in the answer set are the following:

```
contribution("mutual fund AXP",profit,10)
contribution("bond BX",profit,7)
contribution("bond RX",profit,9)
contribution("mutual fund CXP",profit,15)
contribution("mutual fund AXP",availability,11)
contribution("bond BX",availability,13)
contribution("bond RX",availability,11)
contribution("mutual fund CXP",availability,6)
```

```
weight(Username,Dimension,Total):-
  profile(Username,_,_),
  user_score(_,_,Dimension,_),
  Total=#sum{Score,Attribute:
    profile(Username,Attribute,Value),
    user_score(Attribute,Value,Dimension,Score)}.
```

Listing 1.4. Calculation of user weights in ASP

Similarly, all user weights based on the profile and scoring facts are calculated by the ASP rule defined in Listing 1.4. This rule produces the following two solution atoms:

```
weight("Simon",profit,9)
weight("Simon",availability,14)
```

```
utility(Username,Productname,Total):-
  profile(Username,_,_),
  product(Productname,_,_),
  Total=#sum{W*C,Dimension:
    weight(Username,Dimension,W),
    contribution(Productname,Dimension,C)}.
```

Listing 1.5. Utility calculation expressed in ASP

Finally, the rule in Listing 1.5 calculates the resulting utilies based on the product contributions and user weights. The solution atoms produced by the rule in Listing 1.5 are:

```
utility("Simon","mutual fund AXP",244)
utility("Simon","bond BX",245)
utility("Simon","bond RX",235)
utility("Simon","mutual fund CXP",219)
```

At this point we want to emphasize that the three rules depicted in Listing 1.3-1.5, which can be seen as a logic representation of the core of a utility-based recommendation engine, are totally generic and do not have to be changed in case of extending the set of products, product or user attributes, attribute values or scoring values.

Building on such an ASP implementation of a recommendation engine, additional rules can be easily added in order to produce solution atoms for explanations. For instance, if we want to support the top ranked item by a corresponding explanation, we can add the rule in Listing 1.6. This rule basically expresses that a product *Productname* with a utility *U* is top ranked if there is no other product *Productname1* with a higher utility *U1*.

```
top_ranked(Username,Productname,U):-
  utility(Username,Productname,U),
  #count{Productname1:
    utility(Username,Productname1,U1),U1>U}=0.
```

Listing 1.6. Producing an argument for the top ranked product

The rule in Listing 1.6 produces the following solution atom:

```
top_ranked("Simon","bond BX",245)
```

```
top_in_dimension(Dimension,P,C):-
  contribution(P,Dimension,C),
  #count{P1:contribution(P1,Dimension,C1),C1>C}=0.
```

Listing 1.7. Calculating the top item in a dimension

Similarly, we can add a rule for identifying whether a product is best in some dimension. Listing 1.7 shows such a rule, which produces the following atoms:

```
top_in_dimension(profit,"mutual fund CXP",15)
top_in_dimension(availability,"bond BX",13)
```

```
domination(P,"dominated by",P1):-
  product(P,_,_),product(P1,_,_),
  #count{Dimension:contribution(P,Dimension,C),
    contribution(P1,Dimension,C1),C>C1}=0,
  #count{Dimension:contribution(P,Dimension,C),
    contribution(P1,Dimension,C1),C1>C}>=1.
```

Listing 1.8. Calculating totally dominated alternatives

We can also produce counter arguments, for example, on totally dominated product items, i.e. those where there exists an item that is at least equally good in all dimensions and better in at least one dimension. The rule given in Listing 1.8 achieves these *negative* explanations and adds the following atom to the answer set:

```
domination("bond RX","dominated by","mutual fund AXP")
```

5 Conclusions

The paper exemplified a first proof-of-concept for employing the ASP paradigm to develop recommendation applications and lists core predicates and rules. Particularly noteworthy is the ability to efficiently derive explanations including *negative* ones that identify dominated or non Pareto-efficient choice options.

References

1. Anelli, V.W., Basile, P., Bridge, D., Di Noia, T., Lops, P., Musto, C., Narducci, F., Zanker, M.: Knowledge-aware and conversational recommender systems. In: Proceedings of the 12th ACM Conference on Recommender Systems. pp. 521–522. ACM (2018)
2. Dyer, J.S.: Maut multiattribute utility theory. In: Multiple criteria decision analysis: state of the art surveys, pp. 265–292. Springer (2005)
3. Felfernig, A., Teppan, E., Friedrich, G., Isak, K.: Intelligent debugging and repair of utility constraint sets in knowledge-based recommender applications. In: Proceedings of the international conference on intelligent user interfaces (IUI). Springer Berlin Heidelberg
4. Felfernig, A., Friedrich, G., Jannach, D., Zanker, M.: An integrated environment for the development of knowledge-based recommender applications. International Journal of Electronic Commerce **11**(2), 11–34 (2006)
5. Friedrich, G., Zanker, M.: A taxonomy for generating explanations in recommender systems. AI Magazine **32**(3), 90–98 (2011)
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88). pp. 1070 – 1080. MIT Press (1988)
8. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An asp-based system for e-tourism. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 368–381. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
9. Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender systems: an introduction. Cambridge University Press (2010)
10. Leite, J., Ilić, M.: Answer-set programming based dynamic user modeling for recommender systems. In: Neves, J., Santos, M.F., Machado, J.M. (eds.) Progress in Artificial Intelligence. pp. 29–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
11. McNee, S.M., Riedl, J., Konstan, J.A.: Being accurate is not enough: how accuracy metrics have hurt recommender systems. In: CHI'06 extended abstracts on Human factors in computing systems. pp. 1097–1101. ACM (2006)