A Lightweight Ontology for Rating Assessments

Cristiano Longo[1]* and Lorenzo Sciuto[2]

[1] TVBLOB s.r.l. Milano Italy
[2] Università di Catania, Dipartimento di Ingegneria Informatica e delle Telecomunicazioni

Abstract. Various recommender systems and trust engines use application specific formats to store and exchange data. Such data are used for statistical purposes, or to produce recommendations about items or users. This paper introduces **Ratings Ontology**, a semantic web format for ratings and related objects. This ontology aims to be *lightweight*, in the sense of minimising the physical size of data stored or sent over the network for a given ratings set. Moreover, we presents a tools suite based on this ontology to find out statistical information from a ratings data set.

1 Introduction

A big part of on-line services keeps track in different ways of users' taste and satisfaction. Usually this information is collected in form of *ratings* assessed by *users* about *items* of the system itself. *Amazon.com*[1] for example is an on-line book store, that collects user preferences about books. Saved ratings could be used to deduce several kind of statistical information, i.e. to measure user satisfaction about the whole system, or to know best and least liked items. *Recommender Systems* use these ratings to suggest items they may like to the users. As pointed out in [2], recommender system performances increase proportionally to the amount of available information. For this reason, in order to produce good recommendations, it should be convenient to share ratings collected by different, but similarly purposed, on-line services via a common ratings exchange format. Such a format also implies other advantages. It would allow to separate the ratings collection task from the collected data processing, i.e.recommendations production. As a consequence tools and recommender systems could be developed independently from ratings collection engines.

This paper introduces **Ratings Ontology**, a Semantic Web format to store and share ratings. This ontology aims to be *lightweight*, in the sense of minimising the physical size of data stored or sent over the network for a given ratings set. The rest of this paper is structured as follows : Section 2 contains a brief description of semantic web intent and languages; Section 3 describes two ontologies with similar intents to ours, but with more limitations; Section 4 describes the features provided by ratings ontology; Section 5 describes a use-case of this ontology by introducing some tools we developed which are able to process data sets in this format.

* Thanks to G. Di Blasi, P.Oliveto and B.Vintrici for their lexical contribution.

2 Semantic Web Ontologies

Semantic Web provides a common framework to share and reuse data across applications. It provides languages to express information in a machine processable way. The Semantic Web core language is **RDF**[3]. An rdf document can be seen as a graph in which nodes are linked to each other by *properties*. Nodes and properties can be labelled by a Uniform Resource Identifier(URI)[4]. An rdf graph can be seen as a set of triples $(source, property, target)$, that corresponds to graph edges. Such a triple represents a relation identified by *property* that goes from *source* to *target*. The big part of rdf storage engines uses such triples as internal representation of rdf graphs. So the number of triples can be used as a metric to measure the *size* of an rdf document. The **RDF Vocabulary Description Language(RDF Schema)**[5] is a semantic extension of RDF. It provides mechanisms for describing groups of related resources(RDFS classes) and the relationship between them. It allows to define *vocabularies* in terms of classes and properties. **Web Ontology Language(OWL)**[6] enriches the RDF Schema with various constructs and constraints for properties and classes. It defines also some meta-level properties to describe relations between properties and classes. As an example, given two properties p and q, we can state that p is the inverse of q via the $owl : inverseOf$ property. OWL also allows to define cardinality and value constraints, useful to check if instances of a class are consistent; i.e. we can say that a boy has at most one father using $owl : minCardinality$.

3 Related Works

Trust Ontology[7] is an extension of the Friend Of A Friend ontology[8], that defines properties about user profiles. Trust Ontology adds features for *user-to-user* trust assessments. It provides eleven properties, one for each trust value in a zero to ten scale. As a result, trust information is stored in a very compact way into an rdf graph because for each trust assertion just one triple is stored. On the other hand, this ontology allows to express only ratings in a zero to nine scale, and offers no capabilities for other rating spaces. For example, *Movie Lens*[9] uses a five point scale, so Trust Ontology is not suitable for ratings collected by this engine.

Review Ontology[10] has more power. For each rating assessment a corresponding *Review* is defined, with a *rating* and two properties to describe the ratings range: *maxRating* and *minRating*. This ontology suits all systems with discrete finite equispaced rating ranges. However, it is not yet enough, because some engines, i.e. *Moleskiing*[2], allow users to enter ratings in a continuous interval. Moreover, the presence of *maxRating* and *minRating* for each rating entered by a unique engine is redundant and it causes a growth of data storage size.

Ratings Ontology aims to cover the entire ratings spectrum and, at the same time, to reduce the amount of data stored for a given set of ratings.

4 Ratings Ontology

Ratings ontology is an OWL based format that provides classes and properties to represent in an exhaustive and machine processable way ratings collected by web sites, automated agents and other engines. The Ratings Ontology specification is available at [11]. Ratings collected by different engines could be saved in the same storage, in order to increase the accuracy of recommendations, or to get statistics from a larger data set. On the other hand, the engine that collected a rating and the context in which this rating was produced is an important piece of information. For this reason, our ontology provides features to bind a rating with the engine that collected it, and to describe *how* this rating has been collected. The next sections describe the classes and properties introduced by the Ratings Ontology.

4.1 Rating Class

A rating represents a sort of preference, or judgement, *assessed by* a user or a software agent *about* a generic item. For such ratings, Ratings Ontology provides the *Rating* class. Rated items are expressed in terms of RDF resources. This allows to provide a full description of rated items using suitable elements from other ontologies. The rating asserter must be an *Agent*, where the *Agent* class is defined in the *foaf*[8] ontology. We chose the *Agent* class instead of the more restrictive *Person* to cover situations in which a rating assessment was not caused, directly or not, by a physical person, but by an intelligent agent. As an example, [12] describes how trust assessments among grid nodes could be used to improve the performance of the whole computational grid. Attention should be paid for understanding that this is not the case of ratings collected by an automated agent that *measures* in some way how much a user likes an item, i.e. a browser that keeps track of the amount of time you spent on a web page. In such a situation we say that the rating was collected in an *IMPLICIT* way and the person whose behaviour has been observed to produce the rating should be considered as the asserter of the rating itself.

A rating can have a *value*. It is an additional information, whose interpretation depends on the context in which the rating has been produced. The browser of the previous example could value ratings by counting the number of times a user visited a certain web page. The great part of engines that collect *explicit ratings* ask the user to enter a preference about an item. In this case, the *value* property is appropriate to store such a preference. To assure processability and uniformity for third parties software, a rating value must be *numerical*, in the sense that it must be a typed literal with a numeric data type. In a context where no values are assigned to ratings, a rating should be considered as a positive assertion, but the the absence of a rating should be considered like an unknown value, and not as a negative assertion. The Following section contains some code fragments as usage examples of the Ratings Ontology classes and properties. In order to improve readability, we decided to use entity references instead of full name-spaces in URIs. We use *rat* as a shortcut for the ratings ontology name-space, and *xsd* for the XML Schema one.

4.2 Ratings Collection Engines

Information about the rating context and the engine responsible for the collection is encoded by the *RatingsCollector* class instances. It is appropriate to create an instance for each engine, in order to keep track of who is responsible for the collection of a rating. For example, if two web sites use the same software to produce and store ratings, they should be represented by two distinct *RatingsCollector* instances. For each rating collector the *mode* in which this engine works has to be specified. In the $EXPLICIT$ mode the asserter is explicitly asked to enter a rating about a resource, i.e. by using a form. The $IMPLICIT$ mode was introduced in Sect.4.1, and it covers all scenarios in which the user is not asked explicitly to assess a rating, but ratings are produced observing her behaviour. The following code fragment is the definition of a ratings collector that works in explicit mode.

```
<rat:RatingsCollector rdf:about="http://example.collector1.org">
  <rat:mode rdf:resource="&rat;EXPLICIT_MODE" />
  ....
</rat:RatingsCollector>
```

The most common way to collect explicit ratings is to ask users to choose a preference value for an item from a set of available ratings. For example, at the end of a movie, the user could be asked to choose a rating in the set $\{GOOD, BAD\}$. As pointed out in Sec.4.1, these two values must be saved into our rating data base as numerical values, i.e. using 1 for *GOOD* and 0 for BAD. The set of available ratings can vary for each collection engine. For example *MovieLens*[9] allows users to enter preferences in the range from 1 to 5 stars. Moreover, there is some system in which available rating values are not a discrete finite set, as in previous examples, but a continuous interval(*Moleskiing*[2]). Ratings ontology provides two different ways to define the range of available ratings. The first one models a discrete finite ratings set via **exhaustive enumeration** of available rating values. For this purpose Ratings Ontology offers the property *hasAllowedRatingValue*, that allows to specify one by one allowed rating values. The following code fragment shows how to encode a Ratings Collector with mode set to explicit and $1, 2, 3$ as available rating values.

```
<rat:RatingsCollector rdf:about="http://example.collector2.org">
  <rat:mode rdf:resource="&rat;EXPLICIT_MODE" />
  <rat:hasAllowedRatingValue rdf:datatype="&xds;integer">
    1
  </rat:hasAllowedRatingValue>
  <rat:hasAllowedRatingValue rdf:datatype="&xds;integer">
    2
  </rat:hasAllowedRatingValue>
  <rat:hasAllowedRatingValue rdf:datatype="&xds;integer">
    3
  </rat:hasAllowedRatingValue>
</rat:RatingsCollector>
```

The second mechanism is more general but less expressive. At first an interval can be *bounded* or *unbounded*, in one or both directions. The *hasRatingValuesRangeLowerBound* and *hasRatingValuesRangeUpperBound* properties respectively allow to define an upper and a lower bound for a ratings range. The following code fragment shows how to encode intervals $[5, +\infty[\subset \mathbb{R}$ and $[1, 1.5] \subset \mathbb{R}$.

```
<rat:RatingsCollector rdf:about="http://example.collector3.org">
  <rat:mode rdf:resource="&rat;EXPLICIT_MODE />
  <rat:hasRatingValuesRangeLowerBound rdf:datatype="&xds;integer">
    5
  </rat:hasRatingValuesRangeLowerBound>
</rat:RatingsCollector>

<rat:RatingsCollector rdf:about="http://example.collector4.org">
  <rat:mode rdf:resource="&rat;IMPLICIT_MODE />
  <rat:hasRatingValuesRangeLowerBound rdf:datatype="&xds;integer">
    1
  </rat:hasRatingsLowerRangeBound>
  <rat:hasRatingValuesRangeUpperBound rdf:datatype="&xds;float">
    1.5
  </rat:hasRatingValuesRangeUpperBound>
</rat:RatingsCollector>
```

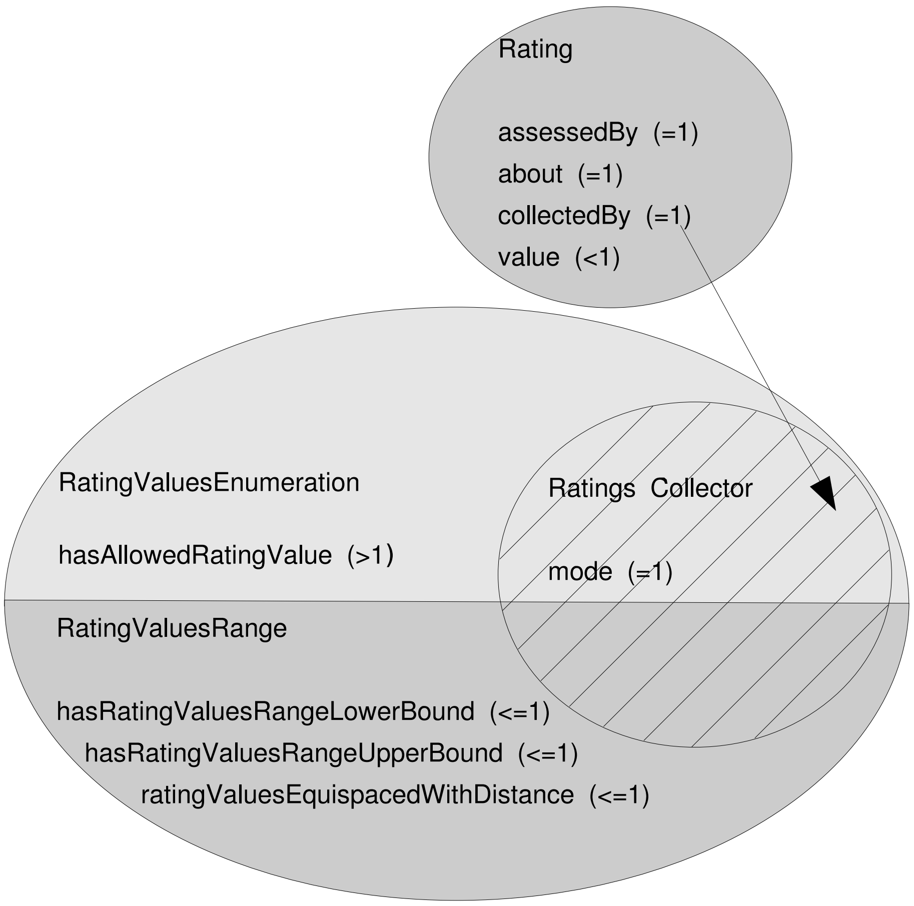
A range defined in this manner is assumed to be continuous. If we have a finite or not finite ratings range, in which available values are equally spaced we can encode it with the *ratingsEquispacedWithDistance* property. Obviously, such a range consists of a set of discrete values. The following code fragment shows how to define the set of even numbers as the ratings range.

```
<rat:RatingsCollector rdf:about="http://example.collector5.org">
  <rat:mode rdf:resource="&rat;EXPLICIT_MODE />
  <rat:hasRatingValuesRangeLowerBound rdf:datatype="&xds;integer">
    0
  </rat:hasRatingValuesRangeLowerBound>
  <rat:ratingsEquispacedWithDistance rdf:datatype="&xds;integer">
    2
  </rat:ratingsEquispacedWithDistance>
</rat:RatingsCollector>
```

Please note that a discrete finite set of equally spaced available ratings could be defined using both of these two mechanisms. In order to increase readability and minimise the storage size, the definition via enumeration should be used only when the amount of available values is not too large.

4.3 Ratings Collector Class Diagram

These two ways to define the set of available ratings are mutually exclusive, so you can't mix them to create an hybrid ratings range. This constraint was made explicit in the ontology definition creating two disjointed classes for rating collectors, one for each range definition mechanism. The *RatingsCollector* class is defined as the union of these two classes, with the additional *mode* property. Figure 1 outlines ratings ontology classes and properties.



Rating
assessedBy (=1)
about (=1)
collectedBy (=1)
value (<1)
RatingValuesEnumeration
hasAllowedRatingValue (>1)
Ratings Collector
mode (=1)
RatingValuesRange
hasRatingValuesRangeLowerBound (<=1)
hasRatingValuesRangeUpperBound (<=1)
ratingValuesEquispacedWithDistance (<=1)

Fig. 1. Ratings Ontology class diagram

4.4 Invalid Ratings

OWL provides features to include into the ontology definition the most of the constraints needed for rating data sets. In example cardinality constraints are used to specify that a rating must have just one asserter. The only additional constraint needed is about rating values related to the set of available ratings provided by the engine. If we found a rating whose value is not allowed by the engine that collected it, this rating should be considered invalid and discarded by tools that process the data set this rating belongs to. The presence of such a rating in the data set probably was caused by an error during the collection task, or by some other processing of the data set itself.

On the other hand, you can define a ratings collector with a ratings range lower bound greater than the upper bound, producing a meaningless definition. We decide to leave unspecified how to handle such a situation, delegating this task to implementations.

5 Applications : Data Set Statistics

As pointed out in Sect.1, a universal format to deal with ratings, as Ratings Ontology aims to become, allowed us to develop software tools that process ratings independently from the engine which collected them. In this section we introduce a set of tools able to find out various statistical information from a set of ratings, stored via the ratings ontology. These tools have been developed using the Jena[13] api for RDF and OWL processing, and the SPARQL[14] support provided by the ARQ engine for queries. The tools are available as set of api together with the command line tools based on the api itself. We have planned to release a *graphical* version in the near future and to make the api also available as a web service.

5.1 Validity Checker

The first tool performs the rating validation described in Sect.4.4. Given a resource into an rdf *model*, the first feature is to detect whether it is a valid rating or not. Moreover we provide a Jena reasoner to discover all errors in a model, signalling also invalid ratings.

5.2 Collection Engines

As pointed out in Sect.4, a common data set could be used to store ratings collected by different engines. The second tool extracts all ratings engines defined into an rdf model, providing also basic information about them like the mode and the set of available ratings.

5.3 Asserters and Items

Our suite also provides features to retrieve the following information from a rating data set :

1. number of rating asserters;
2. number of rated items;
3. the list of all asserters;
4. the list of all rated items;
5. total number of ratings;
6. total number of *distinct* ratings;
7. ratings *density.*

It can happen that a user assesses two different ratings for the same item at two different times. The System that keeps track of this fact should store additional information to distinguish these two different events (for example a timestamp). We say that two ratings are *distinct* if they differ for asserter, rated item or both.

Density measures the amount of available information provided by a ratings set. It coincides with the density of the bi-dimensional matrix labelled with asserters and items, and with rating values into cells. We use the formulation of density that can be found in [15]:

$$Density = \frac{IU}{I * U} \tag{1}$$

where U is the total number of asserters, I is the total number of rated items, and IU is the total number of distinct ratings. All of these calculations can be restricted to a single collection engine. So, given a collection engine E we can retrieve the number of asserters which have at least one rating collected by E, the number of items with at least one rating collected by E, and so on.

5.4 Statistics on rating values

We can find out various statistics from a set of ratings collected by the same, well known, engine. For example we can get:

1. frequency distribution of available rating values;
2. average and variance of ratings;
3. average rating for an item;
4. the list of more rated items.

Dealing with ratings collected by different engines is a more subtle task, because the *raw* numerical value of a rating is meaningless without any information about its collector ratings range. Given two different collector engines $E1$ and $E2$ with a limited ratings range(a rating range with an upper and a lower bound), ratings collected by these two engines could be *normalised* into the interval $[0, 1]$ in order to be processed in a uniform way. For this reason, our suite allows to find out statistics about ratings collected by two or more engines with finite rating ranges.

6 Conclusions And Future Works

This paper introduces Ratings Ontology, an OWL based format to deal with ratings and related matters. This ontology also provides elements to describe collection contexts, that contain all information needed to give a correct interpretation of a rating and of its value. In Sect.5 we presented some tools that work on documents in this format, showing how a uniform exchange format could be useful to develop collection engine independent tools. We have planned to deliver two different applications that deal with data sets with elements described through Ratings Ontology. At first, to help researchers and companies we want to develop a full test suite for collaborative filtering and trust algorithms. This task involves the definition of a *common format* to describe test results, and the development of some tools for statistical results visualisation. Another application of this ontology could be a generic framework for recommender systems. We are considering to use SWAMI[16] as starting point.

References

1. Linden, G., Smith, B., York, J.: Amazon.com recommendations: item-to-item collaborative filtering. In: Internet Computing. Volume 7., IEEE (2003) 76– 80
2. Avesani, P., Massa, P., Tiella, R.: A trust-enhanced recommender system application: Moleskiing. In: Proceedings of the 2005 ACM symposium on Applied computing SAC '05. (2005)
3. Herman, I., Swick, R., Brickley, D.: Resource description framework (rdf) (2004) http://www.w3.org/RDF/.
4. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifier (uri): Generic syntax. In: Request For Comments. Number 3986. IETF
5. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. (2004) http://www.w3.org/TR/rdf-schema/.
6. McGuinness, D.L., van Harmelen, F.: Owl web ontology language overview (2004) http://www.w3.org/TR/owl-features/.
7. Golbeck, J.: The Trust Ontology. http://trust.mindswap.org/trustOnt.shtml.
8. Brickley, D., Miller, L.: The friend of a friend (foaf) project. (2007) http://www.foaf-project.org/.
9. Miller, B., Albert, I., Lam, S., Konstan, J., Riedl, J.: Movielens unplugged: Experiences with a recommender system on four mobile devices. In: 17th Annual Human-Computer Interaction Conference, GroupLens Research (2003) http://movielens.umn.edu/.
10. Ayers, D.: Review vocabulary http://dannyayers.com/xmlns/rev/.
11. Longo, C.: Ratings ontology http://www.tvblob.com/ratings/.
12. Farag, A., Muthucumaru, M.: Evolving and managing trust in grid computing systems. In: Canadian Conference on Electrical Computer Engineering. (2002)
13. McBride, B.: Jena: Implementing the RDF Model and Syntax Specification. In: Semantic Web Workshop, WWW2001. http://jena.sourceforge.net/.
14. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (2007) http://www.w3.org/TR/rdf-sparql-query/.

15. Caldern-Benavides, M.L., Gonzlez-Caro, C.N., de J. Prez-Alczar, J., Garca-Daz, J.C., Delgado, J.: A comparison of several predictive algorithms for collaborative filtering on multi-valued ratings. In: ACM symposium on Applied computing. (2004)
16. Fisher, D., Hildrum, K., Hong, J., Newman, M., Thomas, M., Vuduc, R.: SWAMI: a framework for collaborative filtering algorithm developmen and evaluation. In: Research and Development in Information Retrieval. (2000) 366–368