INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Semantic Recommender System

WILLIAM KINAAN

Outubro de 2016

POLITÉCNICO
DO PORTO

Semantic Recommender

Recommendation Systems

William Kinaan

ABSTRACT

Though Content-based recommender systems proved to have better quality than Collaborative Filtering recommenders, the later is more used because the former suffers from complex mathematical calculations and inadequate data modeling techniques. Using Ontology(ies) to model the data allows machines to better understand both items and users' preferences and thus not just suggesting better recommendations, but also providing accurate justifications.

In this work we present a Semantic Recommender system that uses a novel way of generating recommendations depending on a Recommender Ontology that provides controlled vocabularies in the context of recommendations, and that is built upon the idea that not all classes and properties are important from item-similarities point of view. If the domain Ontology is annotated with the Recommender Ontology, the Semantic Recommender should be able to generate recommendations. As a result, the proposed system works with any domain data. Thanks to The Semantic Web standards.

The proposed mathematical model takes into consideration, in addition to items' features and users' profiles, the context of the users and the temporal context, so some items, as an event's ticket, should never be recommended if the event is over, and should get more presence before the event.

The Recommender Ontology grants business owners a way to boost the recommended items according to their needs. This guarantees more diversity, which satisfies the business requirements.

For the experiments, we have tested the proposed solution with many domains including movies, books, music, and with a real business company. We got 55% accuracy when testing on a movie domain though we knew just one feature about the movies. The main limitation we have faced is the absent of a content-based domain case that contains ABox, TBox, and ratings together.

Keywords: Semantic Recommender, Recommendation Systems, The Semantic Web, Ontology.

ACKNOWLEDGMENT

I wish to express my gratitude to my advisor, Professor Paulo Maio, without his help, coach, and inspiration, nothing would have done.

I would like also to thank the Global Platform for Syrian Students for granting me the opportunity to study abroad. I can't express enough my thankful to their efforts and support.

In the honor of the Syrian army

إلى شهداءِ الجيش العربي السوري, ألفُ تحيَةٍ و سلام

In the honor of Albert Einstein, Charles Darwin, Aristotle, Galileo Galilei, Isaac Newton, Max Planck, Nicolas Tesla, Marie Curie, Leonardo da Vinci, Carl Sagan, Nicolaus Copernicus, Archimedes, Stephen Hawking, Karl Marx, Ludwig van Beethoven, Giuseppe Verdi, Gioachino Rossini, Pyotr Ilyich Tchaikovsky, Gustav Mahler , Wolfgang Amadeus Mozart, Hector Berlioz, Antonio Vivaldi, Maurice Ravel, Antonín Dvořák, Richard Dawkins, Bob Marley, Umm Kulthum, Abdel Halim Hafez, Sabah Fakhri, and Fayrouz

To Francesco Totti

Table Of content

List of figures

List of tables

List of equations

Acronyms

RS	Recommender System
CB	Content-based RS
CF	Collaborative Filtering
FCP	Fraction of Concordant Pairs
IDF	Inverse Document Frequency
IR	Information Retrieval
MAE	Mean Absolute Error
MAP	Mean Average Precision
MRR	Mean Reciprocal Rank
MSE	Mean Square Error
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFs	Resource Definition Framework Schema
RMSE	Root Mean Squared Error
RS	Recommender System
SRC	Spearman Rank Correlation
TF	Term Frequency
TF-IDF	Term Frequency-Inverse Document
URI	Uniform Resource Identifier
WTF	Who To Follow
WWW	World Wide Web

The one exclusive sign of thorough knowledge is the power of teaching.
Those who know, do. Those that understand, teach.

Aristotle

Just over 100 years after he published his general theory of relativity, scientists have found what Albert Einstein predicted as part of the theory: gravitational waves. Feb 11th, 2016

Swan Lake Oboe music sheet

1 Introduction

1.1 A Bit Of History and motivation

The idea of recommendations did not start with the information technology revolution, but it has deep roots in the creatures' history. Animals leave chemical trails for their fellows to recommend them some activities. This behavior was noticed first by the Ancient Greek, who call it φερω, and which scientifically means pheromone. Pheromones are chemicals capable of acting outside the body of the secreting individual to impact the behavior of the receiving individual[1]. They have many types. For instance, *Trail Pheromones* are being generated to help the others locate some resources, this is popular in insects, especially ants, that indeed leave markers to record their routs as a sign and recommendations for the other ants in order to locate nutritious substances, *Alarm Pheromones* are indications for the others that predators are, or might be, around, and *Epideictic Pheromones* are recommendations from a female to the other females about the correct place to hatch. Moreover, bees help each other finding the food by performing *Waggle Dance* that is a dance by a worker bee and its direction and orientation form recommendations for the others about that place where the food is available [2].

The concept has evolved in Humans due to their superior abilities over other species. In daily life activities, people watch, follow, and talk to other people seeking information –in this context the information is recommendations. People depend upon other people opinions when they explore things for the first time. For instance, a person wants to visit a new city will check what beautiful places other people, who have already visited it, suggest, and the decision anyone would undergo about whether or not to watch a movie relies heavily on the reviews critics have written. This behavior is called nowadays as *Social Navigation*, which is a term that was introduced for the first time by Dourish and Chalmers (1994) as "moving towards a cluster of other people, or selecting objects because others have been examining them" [3]. Examples about Social Navigations are everywhere; a lost person in a strange city would request the correct direction from others. Some people who are attending a music concert would follow the others when the show finishes in order to find the exit gates. In general, Information-seeking has always been a necessity for Human. With the information systems emergence, especially hypermedia, hand-written books become electronic document. These systems provide a way to search for data inside documents, which is basically a text search. The invention of the biggest hypermedia application, the World Wild Web, allows the documents to be linked together, and the Internet allows these documents to be easily retrieved. The document space started to increase incredibly leaving people in an urgent need for a search mechanism. Mathematicians began to adopt and create mathematical theories and models in the information system field resulting in what is known as *Search Engines*. Not surprisingly Google, which started as a search engine, is one of the giant software companies today because it offers what people badly need, the search feature, though this feature is not matured enough since the deployed algorithms, either used by Google or other search engines, are unable to provide semantic search. Alongside with that problem, the amount of information in the Web boosts in way not just people cannot process it, but they do not even know about it, which creates a new challenge, that is building *Recommender Systems*. The spread of Internet to the majority of people inspires business companies to sell their products online, and the number of products becomes fabulously large. As a result, a recommender system for any online store turns into a decisive task that helps customers find what they could not have found without it. Thus, increase the profit. Moreover, giant software companies nowadays depend totally on recommender systems to keep their businesses up and growing. For example, Twitter on 2010 needed so badly a recommender system so they built one called (WTF) [4] in three months living in just one server in order to provide their customers with recommendations. Any recommender system must be able to deal with thousands of millions of users and millions of products. Yet, it must be accurate.

Though the current techniques and algorithms that are being used in recommender systems succeed in delivering good recommendations, they still not mature enough to achieve people fulfillment completely. The disadvantages come from the idea that the modern approaches treat users and products as numbers and bytes, they do not understand correctly the characteristics of the products nor users' preferences, and that is why the current search engines do not perform as required. The reason behind that lies in the way data is model in the Web. The documents are not linked semantically nor does the data. The Semantic Web offers a way to let machines understand the data, not just present it. Semantic agents are capable of inferring new knowledge from already-existing data. It allows the piece of data, which represents Human's knowledge, to be linked all together in what Sir Tim Berners-Lee called "*The Web of Data*" or "*Data Web*". As long as machines can understand the data, there must be a way to reflect that development into the recommender systems. That motivates me to do my thesis. Using The Semantic Web in a recommender system grants us providing more diverse products because the machines can, using correct inference rules, discover similar products, and thus, recommend items from many interesting fields for the users, which is not available in the current approaches out of the box. Moreover, if the machines understand the data, they should be able to give correct and accurate justification about the recommendations. The idea lies in its essence in using Ontology(ies), because they provide a coherent way to model the data, so the machines will not just try to calculate blind algorithm trying to find similar products. Thus, should the products be modeled coherently to a detailed Ontology, machines will infer related products without even searching the whole information repository, unlike the current approach that does this using complex mathematical operations.

1.2 Context

The work is studied and presented using Music Ontology that we have developed specially to present the propose solution and its advantages. The proposed solution should work with any domain that can be described through one or more Ontology(ies). We also tried to apply the system in the case of TIMWE Company, which provides a content platform for content providers in order to facilitate the process of publishing their data. However, the data from TIMWE is not sufficient enough, that is why we have tried to test the proposal with many domains including Books, Movies and Music.

1.3 Objects

The goal of the thesis is to develop a Semantic Recommender that works over any Ontology in order to recommend new content to the users depending on the characteristics of their already liked content. That includes building a coherent Recommender Ontology that takes into considerations all the data-modeling possible scenarios for the domain Ontology. Plus, the work also aims to improve the accuracy of the business measures so the recommendable items will be from diverse areas, rather than just similar items to already-purchased items, which may be accurate from informatics point of view but not from business point of view. That is being done by integrating the demographic information of the users and the time of generating the recommendations in the underlining mathematical model, and by allowing the domain experts to manipulate that model. Moreover, this work intends to let machines provide accurate justifications about the recommendations so users can know the reasons behind the presented recommended items. A critical objective for the work is to constrain the search space for recommendable items to a limited area of items rather than searching the whole information space.

1.4 Adopted Methodology

The proposed solution is to give high priorities for the items that share most of the features with the items the user has already liked. It is built upon the idea that in Ontology, some predicates and some classes are more important for calculating the similarities between items than others, while others are not important at all. The system uses a novel idea in which an item does not necessary suite a user always, but it may be a good recommendation just during a specific amount of time. Furthermore, the system tries to implement the fact that business owners have their own reasons to recommend some items to specific users.

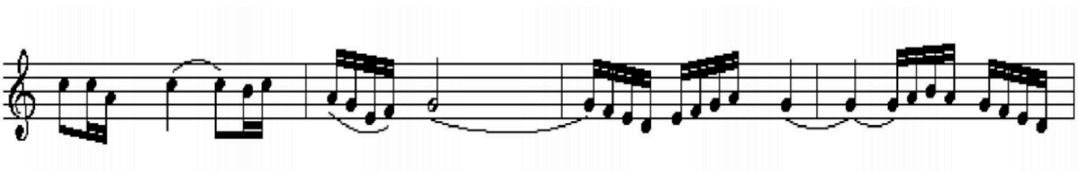
1.5 Structure of the thesis

The thesis is structured in two parts. Part one is the *State of The Art* for recommender systems and a little bit about The Semantic Web and its technologies that will be applied in this work. It contains five chapters[1]. The first chapter is a general view about the recommender systems; its definition, and the aspects that we need to think about when building one. The second chapter describes the Content-Based approach in building a recommender system, items and users modeling, and the way recommendations are being generated. The third chapter discusses the Collaborative Filtering approach to build a recommender system including user-user and item-item approaches. The fourth chapter lists the evaluation measures that have been used to evaluate the quality of a recommender system, such as accuracy metrics, decision support metrics, and ranking metrics. The fifth chapter states a little bit of history from hypermedia to The Semantic Web, and presents the Semantic Web's technologies that we will be using in this research. The second part is about the *Proposed Solution*; it contains three chapters, the first one[2] describes the proposed solutions including the Recommender Ontology, and the recommendation equation. The second chapter describes the implementation of the Semantic Recommender, while in the third chapter we do experiments using many domains such as TIMWE's domain, book's domain, and movie's domain

[1] First chapter in part one is chapter two because the first chapter in this document is an introduction.
[2] First chapter in part two is chapter seven.

It's not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.

Charles Darwin



Bolero violin music sheet

Part I: The State Of The Art

“The true sign of intelligence is not knowledge but imagination.”

Albert Einstein

Peer Gynt Morning flute music sheet

2 Recommender Systems

The aims of this chapter are three-fold:

- First, introduce what Recommender Systems are and the fields they have been used in.
- Second, introduce the aspects that we should think about when building any recommender system.
- Third, provide an abstract architecture for any recommender system.

2.1 Definitions

In literature, many definitions for Recommender Systems (RS) are found, such as:

- Recommendation systems are personalized information filtering technologies used to either predict whether a particular user would like an item or to identify a set of N items that might interest the user [5];
- They are information systems with goal to suggest items of interest to users based on historical behavior of a community of users [6];
- They are systems that collect, store, and process information from users in order to suggest new items for them [7];
- They are intelligent applications that assist users in their information-seeking tasks by suggesting items (products, services, information) that suit the best their needs and preferences [8].

In general, recommender systems suggest new content to the users and/or predict how much they would like an item, depending on analyzing their preferences or other similar users' preferences. In the content of RS, the term *item* is being used interchangeably with the terms *product* and *content*, and it means anything that could be suggested to a user.

To build any RS, there are three main approaches, which are:

1. Content-based Recommender Systems (CB): suggest items that have similar features to the items that the target user has liked in the past.
2. Collaborative Filtering Recommender Systems (CF): recommended items are generated from the ratings that other users have giving to the items not from items' features. CF has two different approaches:
    a. User-User CF: find similar users to the target user depending on the ratings, and then suggest items from these similar users;

b. Item-Item CF: find similar items to the items that the active user[1] has liked, depending on the ratings.

3. Hybrid Techniques.

RS has been use in too many fields, for instance:

- Jokes, such as Jester [2]
- Music, such as Pandora [3], and last.fm [4]
- Movies, such as Netflix [5]
- Academic Papers
- Wine, such as WineGenius [6]
- Social Networks
- Online Shopping, such as Amazon [7] and eBay [8]
- Books

There are many open source libraries to build a recommender system. For instance:

- MyMediaLite[9]
- LensKit[10]
- Duine Framework [11]
- Crab Recommender System[12]
- Recommenderlab[13]

When building any RS, one should consider a diverse of aspects that will define the behavior of it and the way we should build it, section 2.3 explain those aspects. Section 2.2 provides an abstract architecture of any RS.

[1] The Active User is the user that the system is making recommendation for. She/he is also called the Target User.
[2] http://eigentaste.berkeley.edu/
[3] http://www.pandora.com
[4] http://last.fm
[5] https://www.netflix.com
[6] https://www.winegenius.com/
[7] http://www.amazon.com
[8] http://www.ebay.com
[9] http://www.mymedialite.net
[10] http://lenskit.org
[11] http://www.duineframework.org
[12] http://muricoca.github.io/crab/
[13] https://cran.r-project.org/web/packages/recommenderlab/index.html

2.2 Abstract Architecture

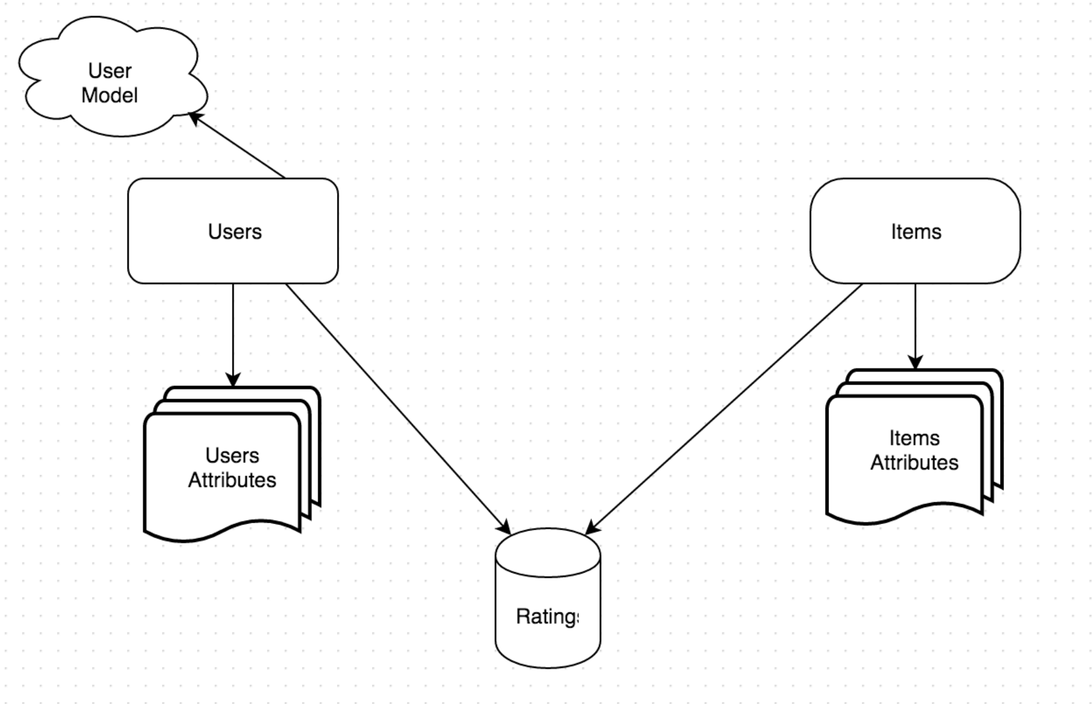


Figure 2-1 the generic architecture for any RS

As illustrated in Figure 2-1,each RS contains [1]

1. A list of users, where each user could be modeled according to many attributes, which we call it *Users' Attributes*. Such as age, gender, or domain attributes, such as *drama* in a movie RS that represents how much this user likes dram movies.
2. A list of items, which are the products that the RS will recommend to the users. These items could be structured in a database tables or they could be text.
3. Items' attributes are the attributes that items are being model according to. For instance, if we are recommending text documents, the items' attributes could be the terms that the systems extracts when it indexes these documents.
4. The Ratings represent users' preferences upon the items.
5. The *User Model* is the model for each user. In CB, the model could be a classification model or a probability model. For instance, a classification model could be built upon two target classes, which are like and dislike. Using the attributes of the items the users have rated in the past, we can build the

[1] These components are the general one. However, in some RS implementations, such as CF User-User, the user model component is not always there.

model using many techniques, such as, decision trees or k-mean classification. Another way to build the model could be using hierarchical clustering or partitional clustering.

2.3 Aspects

To build any RS, there are many aspects that we should think about. These aspects determine the correct approach, and help to select the best evaluation measures.

2.3.1 Domain

The domain represents the context in which RS will work, which will affect the kind of items to be recommended. For instance, in a news domain, RS generates text-formatted recommendations, in a product-sale domain, the recommendations could be just new products, or it could be a bundle that contains other similar/related items but with an offer to the active user, and in a social matching domain, such as Facebook, the recommendations could be other similar users. Moreover, the domain affects the *New/Old items dilemma*, which is trying to answer the question "*Should the RS recommend already-seen items?*" For instance, in a movie context, such as Netflix[1] where the main purpose of the RS is to recommend new movies, the RS should never suggest an already-watched movie. However, in a grocery context, the RS could suggest already-bought products because people usually buy the same items regularly, such as banana, wine, and cheese.

2.3.2 Purpose

The purpose aspect represents the goal of building the RS, for example, increasing the sales and building communities, such as LinkedIn, Facebook, or Tripadvisor. Information seeking is another purposes; such as the RS that recommends scientific papers just to students[2]. Also, some purposes are critical. For example, in the case of Twitter (WTF [4]), where the goal of their RS in 2010 was to keep the system alive since they were facing a decreasing number of users comparing to other social networks.

---

[1] https://www.netflix.com/pt-en/

[2] Just to Oxford University's students

2.3.3 Context

The context is anything apart from the products' ratings that might affect the desirability of particular recommendations at the time of generating the recommendations. For instance:

1. Weather: Only recommend outside activities when the weather is nice.
2. Location: find things nearby.
3. Seasonality: do not recommend winter clothes while we are in the summer, and if it is Christmas, the RS should recommend related items to it.
4. Current activities: are the users shopping now, listening to music, or hanging out with other people? The current activities could constrain/change the behavior of the RS. For instance, if the user is listening to music right now, the RS should not recommend new music in the following way: "Here are 12 songs you may want to consider". It may play the songs directly, or do other things. Moreover, if the system knows the active user is having fun with her/his friends, it may generate recommendations to the whole group rather for an individual. Of course, all this depends on the level of attending the user gives to the RS and the degree in which the user is willing to accept the interrupting.

2.3.4 Who's Opinion?

Opinion aspect cares about the source of the users whose opinions are taking into considerations when generate the recommendations. It is usually one of the followings:

1. Only the active user.
2. Experts: there are RS that use knowledge base, and the recommendations could be manual or automated. The manual case is like the RS of wine.com in which there is the knowledge base is coming from wine experts who would recommend other wines depending on the kind of wine the active user has liked[1].
3. Friends of the active user.
4. Similar people to the active user. Such as Collaborative Filter Item-Item approach described in section 4.3.

[1] Wine.com now uses automatic knowledge-based recommendation system, while in the past; it depended totally on one expert's knowledge.

2.3.5 Personalization Level

The personalization level aspect cares about how personal the RS is. There are many kinds of personalization levels, which are:

1. General (Impersonalized): The same recommendations are being generated for all users. Normally each RS uses an impersonalized version when the user is not logged in. Amazon [1], for instance, shows users the most sold items in a specific context as long as the user is anonymous.
2. Demographic (Personalized), where the demographic information of the user affects the recommendations. For instance, the system should differentiate between the products that it suggests to men and those for women, and the system obviously should not recommend the same items –in most cases- for children, youth, adults and seniors. Also, people in different countries prefer different colors. Thus, it would be a good user-experience if the RS takes this info into consideration [2].

A good example is a study [10] on Trip Advisor that uses a highly demographic algorithm to provide recommendations. Another research showed that the demographics information, such as age and gender, are crucial in evaluating any recommendation system [8]

3. Ephemeral (Personalized): Generating recommendations is done depending on the current activities[3] not the long-term activities. For instance, when a user buys a book, the system could recommend similar books in a users-also-buy section.[4]

2.3.6 Privacy and Trustworthiness

The privacy and trustworthiness aspect cares about privacy and security of the user's info that will be used in one of the recommendations phases. For instance, can the RS use personal information to build models or generate recommendations? Can the user deny some of the preferences? Is the RS honest such as, when the system recommends a hotel to book a

---

[1] https://www.amazon.com/

[2] A study [9] was applied on Netherlands, Japan, Vietnam, and China shows that people in these counties have different color preferences for products such as clothes, personal computers, refrigerators … etc.

[3] It could be the activities that the active user is doing now, or the activities that she/he has done in a specific date range.

[4] This could be thought of as a classification problem as well.

room or an item to be sold, should the system confirm first that the hotel is not full or the store has the item now? And how to handle Shilling Attack scenarios [1]?

2.3.7 Interface

The interface aspect cares about two main characteristics. First of all, the feedback mechanism, which is either explicit or implicit. Secondly, the output of the RS, which could be categorized to one of the followings:

a. Predications: Estimating of how much a user would like an item. It is a numeric score that is supposed to correlate with user's opinion of an item. It is normally scale to match the ratings scales and often tied to searching or browsing for specific product. For instance, when a user checks an item, the RS could tell the user how much it thinks she/he would like this item. Note: In some context, the RS does not show the numeric value because a wrong value could affect the honestly of the RS. For example, if you search on *Google Image* for your picture, the results are a list filtered according to a prediction. However, Google does not show the predicated value because it does not make any sense to say: "Google thinks that this is you by 90%". Figure 2-2 shows a prediction-output example from MovieLens [2].

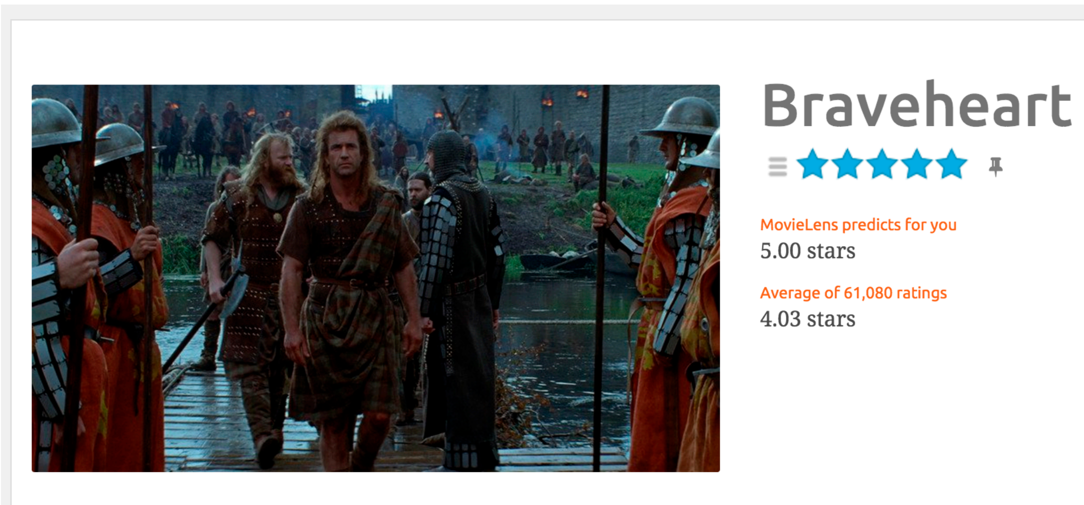


Figure 2-2 Predication-output example from Movielens

b. Recommendations. Suggestions for items users might like. Often they are presented in the form of *Top N List*, and sometimes they are just placed in front

[1] The cases where a user rates her/his products high, while rates the other products low
[2] https://movielens.org/home

of the user. Recommendations are less dangerous to the users from a trustful point of view comparing to the predications, because in the prediction, the RS gives a numeric value to the user, while the user could argue that she/he has bought that item (or watched that movie) and the predication is terribly wrong. Figure 2-3 shows an example of recommendation-output from Amazon [1].

Figure 2-3 Recommendations-output example from Amazon

c. Filtering. If the user searches for something, should the RS filter the results according to what it thinks the user will like more. Figure 2-4 shows a filtering-output example. In this example, we searched for all Charlize Theron's movies.

Figure 2-4 Filtering-output example

2.3.8 Ratings Design

This aspect cares about the way users can express their opinions about the items, which could be done in many ways [11], such as:

1. Unary we know that if someone buys an item then she/he likes it, otherwise, we do not know whether she/he likes it or not.
2. Binary (Thumbs up, Thumbs down)
3. Five star

[1] https://www.amazon.com/

4. 100-point slider

2.3.9 Summary

When building a RS, there are many aspects that we need to think about. They not just affect the chosen algorithm, but the whole system's architecture as well. The chosen approach should balance these aspects and yet takes into consideration the most important ones. Different businesses weight those aspects in many ways, and it is essential to study all of them during the analyze phase since they influence the way RS is built.

There are many approaches to build a recommender system, one of them is the Content-based approach, and that is what the next chapter is about.

“Look again at that dot. That's here. That's home. That's us. The Earth is a very small stage in a vast cosmic arena. It is the only world known so far to harbor life. There is nowhere else, at least in the near future, to which our species could migrate. Visit, yes. Settle, not yet. Like it or not, for the moment the Earth is where we make our stand.”

Carl Sagan

3 Content-based Recommender Systems

The aims of this chapter is three-fold:

1. First, introduction how Content-Based approach works, and what content means in this context.
2. Second, introduce how users and items are modeled.
3. Third, identifying drawbacks, challenges, and benefits.

3.1 Introduction

3.1.1 The idea

Content-Based Recommender Systems (CB) use content that the user has liked in the past in order to suggest similar content. Its techniques are derived mainly from both Information Retrieval field when it comes to extract useful information from unstructured data, and Artificial Intelligence field when it comes to predict new knowledge from previously known knowledge. The generic algorithm to achieve this idea is explained in section 3.1.2. Most of CB share common tasks, such as indexing and TF-IDF calculating. Thus, when building a CB, we can use available tools to achieve those tasks. Some of the available tools are listed in section 3.1.4.

3.1.2 Generic Algorithm

A study [12] shows that any CB follows the following three steps:

1. Model the items according to relevant attributes, as described in section 3.2
2. Model or learn users' preferences according to the same attributes, as described in section 3.2
3. Find a way to calculate the similarities between the items and the users. This step includes building a model for the users and then using that model to find recommendations, as described in section 3.3

The system that achieves this generic algorithm should have a higher level architecture that contains the following components [13], which are described in Figure 3-1:

1. *Content Analyzer* Component. It is responsible of creating structured items out of the information sources by extracting their attributes (features), and represent them according to these attributes. Section 0 shows an example of the output of this component.

2. *Profile Learner* Component. It is responsible of creating users' profiles depending on their interaction with the items. The input for this component is both structured items and user's interactions with these items, which could be data collected from users explicitly or implicitly, or it could be the users' feedbacks on the recommended items). Section 3.1.3 explains more how the RS gets users' preferences, and section 0 shows an output of this component.
3. *Recommender* Component. It is responsible of making the actual recommendation for the active user. Its inputs are the user's profile and the structured items, and its output is either recommendations or predictions.
4. *Feedback* Component. It is responsible of taking the active user's feedback on the recommendable items. The feedback could be implicit or explicit.

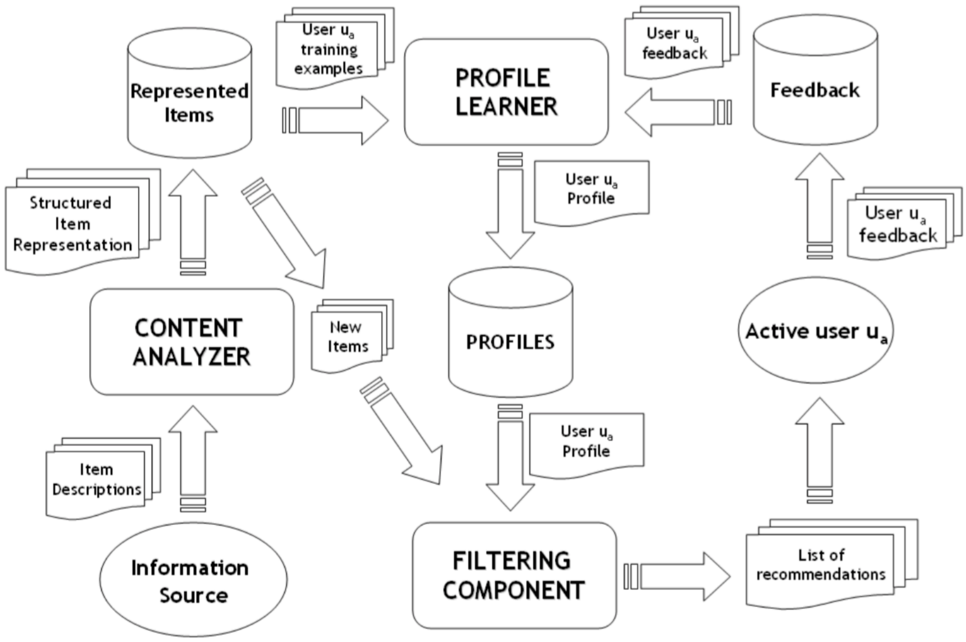


Figure 3-1 High-level architecture of CB Components.

Example: in a news context, the information sources are text articles. The *Content Analyzer* component extracts terms from these articles such as: *Technology*, *Oxford University*, *Cambridge University*, *Restaurants*, *Sport*, *Music*, *Universe*, *Francesco Totti*, and *Physics*. Then the *Content Analyzer* represents the articles according to these terms. For example, Article one is: Technology (90%) and Sport (10%). The *Profile Learner* component will represent the users according to the same terms. For instance, *Van Gogh* is Physics (90%), University (90%), Restaurants (30%), and Francesco Totti (100%).

3.1.3 Building user's preferences

CB needs to know which items users prefer or don't prefer. There are two main ways to collect this information, which are manual and automatic. In the manual way, the system asks the users explicitly to fill their preferences. For instance, the system presents 1000 actors and asks the users to rate them. Moreover, the system may present an interface to the users each time in a while asking them about their opinion in some terms of the system. On the other hand, in the automated way, the system tries to infer users' preferences from their actions. For instance, if the user listens to a specific song many times a day, the system infers that the user likes it.

If the items that the system deals with are unstructured text, RS could use TFIDF (described in annex 10.3) to extract users' preferences.

3.1.4 Available tools

Some of the individual operations for building CB are available and we can reuse them. As mentioned earlier in section 3.1.1, many of CB ideas come from Information Retrieval field of study. Thus, when building a CB, we can take benefit of those tools. For example:

Apache Lucene[1], it is an open source JAVA package that provides document indexing [2], and implements many weighting functions such as TF-IDF. It has implementations in many programming languages. For instance, PyLucene[3] is a Python implementation, Lucy[4] is a C implementation, Lucenenet[5] is a .NET implementation, and SOLR[6] is a REST API on the top of Lucene.

jCOLIBRI[7] is a JAVA case based reasoning platform that can be used if the CF uses Case Based Reasoning techniques.

Weka[8] machine learning software that helps if finding the similarity was based on a classification model, as will be described in section 3.3.2.

---

1 https://lucene.apache.org/core/
2 In the case of CF, the document can be an item.
3 http://lucene.apache.org/pylucene/
4 http://lucy.apache.org/
5 https://lucenenet.apache.org/
6 http://lucene.apache.org/solr/
7 http://gaia.fdi.ucm.es/research/colibri/jcolibri
8 http://www.cs.waikato.ac.nz/ml/weka/

3.2 Item and users modeling

There are many ways to represent both items and users, such as:

1. Vectors in a vector space model, which is described in section 3.2.1
2. Triples using Ontology(ies), which is the adopted way by our proposed solution (cf. chapter 7).

3.2.1 Vector Space Model

In this approach both items and users represented as vectors in a space of terms. The value of an item's vector in each term's dimension is the item's weight against that term [14] [15], which could be calculated using either TF-IDF or using a simple Boolean representation[1], in which an item's weight on a term's dimension either zero or one. The value of a user's vector in each term's dimension is the user's weight against that term, which can be calculated by aggregating the vectors of all the items that the user has expressed her/his opinion on. The aggregation process could be just the *Dot Product* if we normalize the vectors.

3.3 Calculating the similarities

There are many approaches to build the users' models. For example

3.3.1 Cosine similarities in vector space model

The similarities between an item and a user are calculated by cosine the angle between the user's vector and the item's vector. If the two vectors are normalized, we can use the *Dot Product*. For example, in a music context, let us say that we have just two terms, which are *Baroque* and *Classic*. After modeling the items, we got the following structured items:

"*Water Music*" is 80% Baroque and 20% Classic.

"*The Four Season*" item is 10% Baroque and 90% Classic.

The system has modeled the active user, *William*, after analyzing his preferences, as 70% Classic and 30% Baroque. Calculating the cosine between *William's* vector and *Water Music's* vector, and between *William's* vector and *The Four Seasons'* vector, we can suggest The *Four Season* to *William* first. The vector space for this example is illustrated in Figure 3-2

[1] A Boolean representation is not as bad as it sounds. For example, in movies context, a movie is either has Scarlett Johansson actor or not.

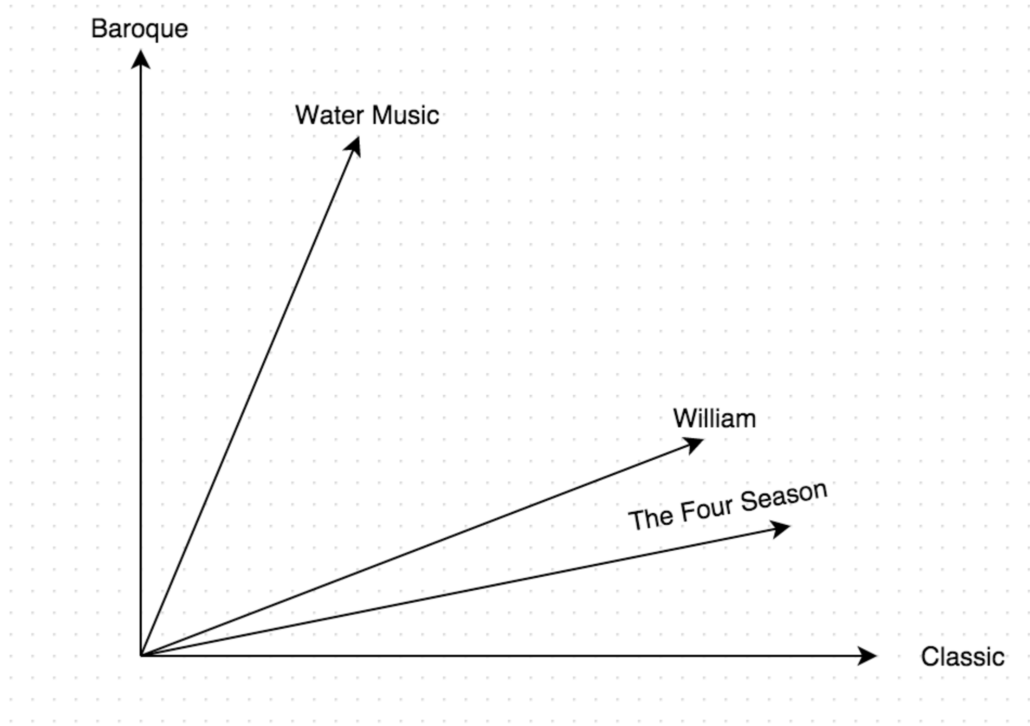


Figure 3-2 Vector space example

Pandora is a music RS that follows the CB approach and uses Vector Space Model to calculate the similarities. The songs[1] are being modeled according to the their musical characteristics. To do that, they started the Music Genome Project [2], which aims to model the songs to 450 different attributes (Music DNA). The job is manually and done by music expert. The algorithm builds a model for the users according to the characteristics of their preferred songs, and then compares these models with the songs in order to find good-and-related recommendations [16].

There are many advantages to use the vector space model; it uses the actual features of the items, which should account for better results, and we do not need a lot of users. One user is enough. On the contrary, the Collaborative Filtering approach needs a big matrix of user's ratings to build the model. Moreover, The computations are relatively easy comparing to other approaches.

However, there are some limitations to use the vector space model; it is a simple model because it can't handle interdependencies, which means that the vector space model does not have the ability to model interaction effects among different dimensions. For

[1] Songs in this context refer to the actual songs, symphonies and all music pieces.
[2] https://www.pandora.com/about/mgp.

instance, a user may like a specific actor in actions movies, and another actor in romance movies, but not vice versa.

3.3.2 Classification Model

Calculating the similarities does not always mean to get a numeric value. There are models that user classification techniques to decide where or the not the user would like an item. The target class is like or dislike. The training set contains the items that we know (either explicitly or implicitly that the user likes or dislikes). *The Daily Learner* system and the *Gixo* system used K Nearest Neighbor classifier, and also Fab [17]. Another ways is Naïve Bayesian classifier [18]

3.3.3 Case Based Model

In this case, the system has already got attributes for the items. However, in the Case based recommenders, the system doesn't recommend items directly to the users, but it asks the user through an interface about what he would like to buy (watch). Then the system checks the attributes for the item the user chooses, and for each attribute, the system checks the available values for it. After that the system asks the users about the preferred values of those attributes in order to reduce the space of possible items that the system thinks that user could like. For instance: if the user chooses laptop, the system checks the attributes for laptops and it gets something like *RAM*, *Screen Size*, and *Processor*. For RAM, the system checks the available values, and if finds 2GB and 4 GB. The system asks the user if he wants a 2GB RAM or 4 GB and so one.

3.4 Summary

3.4.1 CB challenges

1. We need well-structured attributes that align with preferences (think of paintings, it is really hard to describe a painting). It is hard to model the items according to terms or attributes in almost all of the current CB approaches, that is why we have chosen Ontology(ies) to model the items in the proposed solution.
2. We need a good distribution of the attributes across the items and vice versa. (If all the shorts you have are cotton, then cotton won't help)

3.4.2 CB advantages

1. Transparency: the system can explain to the user why it recommends a specific item. It can list the features in the item that they were used to generate the recommendations, while this is not possible in Collaborative Filtering. Nevertheless, the justifications can't be so accurate if the items are not modeled very well. We will see later, in chapter 7, that the proposed solution gives very accurate justifications.
2. If the system has a new item, that item could be recommend the recommendation process doesn't depend on the ratings. However, it is less likely that the new item will be available to be recommended exactly when it is being inserted to the system because in many approaches the model will be calculated each week or even more. The proposed solution solves this problem as well. On the other hand, CF approaches suffer from that problem.

3.4.3 CB disadvantages

1. User Independence: It does not take benefit of similar users.
2. The way of extracting structured data from unstructured data is not sufficient enough.
3. No unexpected items. The items that will be recommended are mainly similar to those that the user has expressed his opinion on previously. Thus, the system will not recommend anything novel. This problem, however, is solved in the Collaborative Filtering approach.

The disadvantages of CB motivate to use another approach in building RS, which is the Collaborative Filtering approach, and that is what the next chapter is about.

Vivaldi’s Spring Music Score

Beethoven’s 5th symphony’s famous four-note

Beethoven’s 9th symphony, one of the best pieces mankind has ever composed.

“Music is … A higher revelation than all Wisdom & Philosophy”

Ludwig van Beethoven

4 Collaborative Filtering Recommender Systems

The aims of this chapter are four-fold:

1. First, provide a history about this approach, and the main two algorithms for it.
2. Second, identify and describe the challenges that this approach faces.
3. Third, describe item-item CF
4. Fourth, describe user-user CF

4.1 Introduction

Humans used collaborative filtering since the beginning of the history. For example, while discovering fruits, if a man is hungry and he found a new fruit, he would check what happened to the other people who have eaten it in order to decide if it is healthy or poisonous. Animals, as well, use collaborative filtering techniques all the time. For instance, ants follow each other on the exact same road in order to find food. The first time the term Collaborative Filtering was mentioned is on Tapestry system [19] on 1992, in which people could query the actions of others. Some of the early works on Collaborative Filtering were published between 1994 and 1995; they used User-User approach, which will be described later on section 4.2, such as:

- GroupLens: it is a recommender system for news
- Ringo/HOMR: it is a recommender system for music, came from MIT Media Lap.
- Video Recommender.

All CF share the same mathematical model, which is presented in the next section.

4.1.1 Mathematical Model

According to [20], the model for any CF recommender system is as the followings:

There is a list of m users $U = \{u_1, u_2 \dots , u_m\}$

There is a list of n items $I = \{i_1, i_2 \dots , i_n\}$

Each user u_i has a list $I_{u\,i} \subseteq I$, which contains the items that this user has already rated.

The active user $u_a \in U$ is the user that the RS is either making prediction or recommendation for.

Prediction $P_{a,j}$ is a numeric value that represents how much the system thinks the active user u_a would like/dislike the item i_j where $i_j \notin I_{u\,a}$

Recommendation $I_r \subset I$ is a list of N Items represents the items that the system thinks the active user u_a is interested in, and where $I_r \cap I_{u\,a} = \emptyset$

R is $m \times n$ user-item matrix where each entry $a_{i\,,j}$ represents the rating of the ith user on the jth item.

To build an information system depending on this mathematical model, there are two main approaches, listed in section 4.1.2. However, there are also challenges we will face during building any CF, some of them are listed in section 4.1.3

4.1.2 Approach

In CF, there are two approaches:

1. User-User Collaborative Filtering [1]. It generates recommendations depending on the items that other users similar to the active user have liked. It's described in more depth in section 4.2
2. Item-Item Collaborative Filtering [2]. It generates recommendations depending on the items that are similar to the ones the active user has liked. It's described in more depth in section 4.3

4.1.3 CF Challenges

The main challenges for any Collaborative Filtering system are [21] :

1. Data Sparsity

The matrix of user-item ratings are almost zeros because normally the number of both the items and the users is very large and the number of items that any user rates is relatively small. This could lead to the following problems:

1. *Cold Start* Problem: Users are unlikely to be given good recommendations when they enter the system for the first time because of the lack of their history. A research [22] tried to solve the cold start problem by using system-controlled techniques for learning user profiles. Most of the times, RS bypass this problem by asking the use explicitly to choose his preferences, Figure 4-1 shows an example from *AllMusic*[3] website.

[1] It has other synonyms such as *User-based* and *memory-based*
[2] It has other synonyms such as item-based and model-based
[3] http://www.allmusic.com/

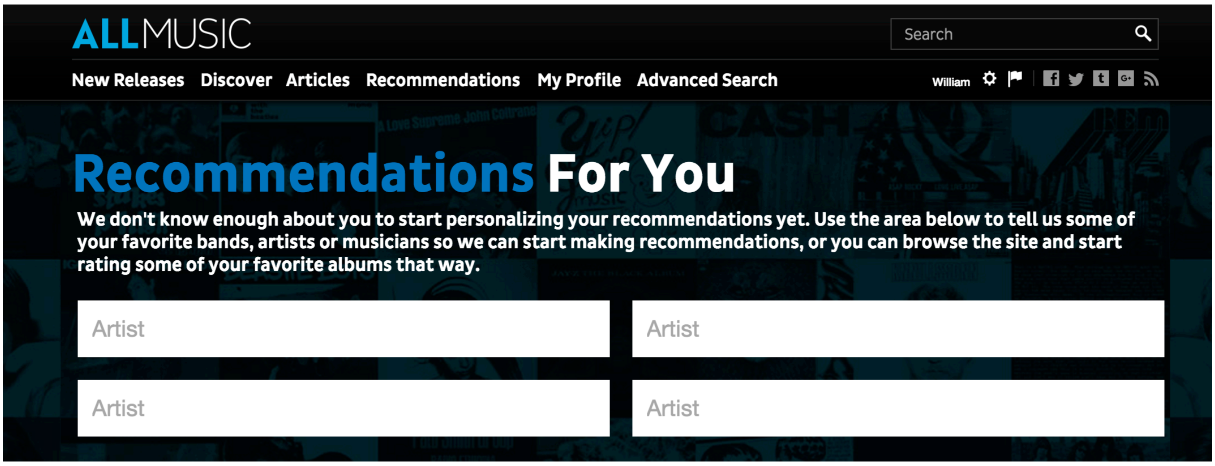


Figure 4-1 Cold start problem example

2. *The Reduced Coverage* Problem: The number of items that the system can recommend is small comparing to the large number of items the system has. This problem occurs when the users do not rate too many items. Thus, the system does not know their preferences. A proposed solution is Filterbots [6] , in which the system fills the mission values depending on many characters.
3. The *Neighbor Transitivity* problem: the system is not able to find similar users unless they have rated the same items. At the beginning, users would not have rated too many items. Thus, the system will not be recognizing them as similar.

2. Scalability

When the number of users and items become big, computations will definitely affect the performance. A good solution is to use *Singular Value Decomposition* (SVD). SVD is a matrix factorization technique that takes a $m \times n$ matrix and produces, according to Equation 1, the following three matrices:

1. **S:** $m \times n$ diagonal matrix with non negative numbers
2. **U:** $m \times m$ orthogonal matrix
3. **V:** $n \times n$ orthogonal matrix

Equation 1 SVD equation

$$M = U \cdot S \cdot V^T$$

Examples of SVD is provided in the annex section 10.2.

However, that creates a new problem, which is the complexity of matrix factorization, which could be solved by incremental SVD pre computation [23] , that makes a pre calculation for the matrices and then when a new users comes, it uses folding-in projection techniques for updating them [24]

3. Gray Sheep

They are the users whose taste do not agree nor disagree with the others. A possible solution to solve it was a proposal to use a hybrid system of Content-based and Collaborative Filtering approaches [25]

4. Black Sheep

They are the users whose taste makes it impossible to make recommendation for them, even a manual RS would not help in this case.

5. Shilling Attack

Where people provide high ratings for their items, and lower ratings for the other items (usually competitors' items). A study [26] shows that Item-Item CF is less preventing a shilling attack than User-User CF.

6. Privacy

People may not wish to let other people know that they are interested in some items. A study [7] suggested a way to protect privacy but that may lead to an uncertainty in the recommendations. It is a tradeoff between privacy and accuracy. The system was tested on Netflix dataset, and the results were somehow good.

4.2 User-User CF

4.2.1 Assumption

If a user X and a user Y have liked 10 items, then if Y likes a new item, X is likely to like it as well. In other words: out past agreement predicts out future agreements. This assumption leads User-User CF systems to share similar characteristics, as described in section 4.2.2. They are also end up using a generic algorithm, as listed in section 4.2.3. However, they all suffer the most from extremely heavy computations calculations problem, which is listed with some suggestions to solve it in section 4.2.4

4.2.2 Characteristics

All User-User CF systems share the following characteristics:

- Ratings: they all provide a way to allow users to rate the items.
- Users' Similarities: they all calculate the similarities between users using different methods. For instance, correlation and vector cosine.
- Personalized Recommendations / Predictions: they all provide recommendations and predictions depending on weighting combinations of other users' ratings.

4.2.3 Generic Algorithm

All User-User Collaborative Filtering share the following steps: [27]:

1. Calculate the similarities between users and save that as the weight between them. This step is to find the neighborhood of the active user. Some various methods to calculate the similarities are listed in section 4.2.5.
2. Select the most k similar users to the active user. The k can be the number of users that their similarity to the active user is bigger than a specific threshold[1].
3. Prediction and Recommendation Computations. In this step, the system computes predications from a weighted combination of the selected neighbors' ratings. Some various methods to do that are listed in section 4.2.6

4.2.4 Computation Problem

All User-User CF systems face calculations (computations) problems, which is: In a system that has m users and n items:

- Creating a correlations between two users is a $O(n)$ problem[2];
- Creating correlations for one user is a $O(mn)$ problem[3]; and
- Creating correlations for all the users is a $O\ (m^2n)$ problem.

Solutions for Computations Problems:

Finding the similarities between users is the bottleneck in this approach [28].Since the number of users becomes millions and the number of items becomes extremely large, User-

[1] Some papers would add another step here, which is normalizing the data. However, normalizing the data is already done in the equations of the third step, that is we have assumed that normalizing the data is already embedded in the third step.
[2] Because we need to check all the items as will be described in User-User and Item-Item approaches in the coming sections.
[3] Because it is $O(n)$ to calculate the correlations for one user with another one. Thus, if we have m users, the complexity will be $O(mn)$.

User Collaborative Filtering becomes inefficient, which leads researchers to create new approaches, which provide better performance and either close accuracy results of even better in some cases. For instance:

- Item-Item Collaborative Filtering.
- Dimension Reduction Techniques.
- Cache the neighbors for users.
- Choose k-size neighborhood, where k is way much smaller than m

4.2.5 Similarity Computations[1]

There are many ways to calculate the similarity between two uses. Some of them are:

1. Correlations-based Similarities:

According to: [21] [27]

Equation 2 Correlations-based similarity equation

$$w_{u,v} = \frac{\sum_{i \in I}(r_{u,i} - \overline{r_u})(r_{u,i} - \overline{r_v})}{\sqrt{\sum_{i \in I}(r_{u,i} - \overline{r_u})^2} \sqrt{\sum_{i \in I}(r_{v,i} - \overline{r_v})^2}}$$

Where $w_{u,v}$ is the weight (similarity) between the user u and the user v, $r_{u,i}$ is the rating of the user u to the item i, and $\overline{r_u}$ is the average ratings of the user u[2].

2. Vector Cosine Similarities:

Like in the Information Retrieval area, we represent any document as a vector in a space, which its dimensions are the terms in the corpus, here we can represent any user as a vector in an item-space, and instead of using the frequency of the term, we use the ratings. [29]

Equation 3 Vector Cosine similarity equation

$$w_{u,v} = \cos(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| * \|\vec{v}\|}$$

[1] In some papers, this step is called *Finding Users Neighborhoods*.

[2] We calculate $\overline{r}_u$because some users prefer to rate just the excellent items, from their point of view, as 5 stars, while others rate the good items ,from their point of view, as 5 stars.

4.2.6 Predication and Computations

There are many ways to compute the recommendations after getting the similar users. Some of them are:

1. ***Weighted Ratings of Others' Ratings:***

Equation 4 Weighted Ratings Predication equation

$$P_{a,i} = \bar{r}_a + \frac{\sum_{u \in U}(r_{u,i} - \bar{r}_u) \cdot w_{a,u}}{\sum_{u \in U}|w_{a,u}|}$$

Where $P_{a,i}$ is the predication of how much the user u will like the item i, $w_{a,u}$ is the weight (similarity) between the user u and the user a, U is the neighborhood of the active user a

2. ***Simple Weighted Average:***

Equation 5 Simple Weighted Average equation

$$P_{u,i} = \frac{\sum_{n \in N} r_{u,n} \, w_{i,n}}{\sum_{n \in N}|w_{i,n}|}$$

4.2.7 Example

Let us say we have a system where users can buy and rate movies. Ratings range from 0 until 7, where 7 represents that the user likes the movie the most, and 0 states that the user does not like the move. We got the following ratings from the users:

Table 4-1 Sample data for User-User CF

	A Beautiful Mind	Legends of The Fall	Godfather 1	Harry Potter
Sarah	7	6	3	7
Elizabeth	7	4	4	6
Maria	3	7	7	2
Suzan	4	4	6	2

We got a new user, William, who has already rated three movies, as illustrated in Table 4-2, and we need to know if we should recommend the fourth movie for him.

Table 4-2 Sample data 2 for User-User CF

	A Beautiful Mind	Legends of The Fall	Godfather 1	Harry Potter
William	7	4	3	**?**

First, we need to calculate the similarities between William and the other users. We will use Pearson Correlation [21] [27], which is the Correlation-based similarity measure, illustrated in section 4.2.5.

William with Sarah	0.85
William with Elizabeth	0.97
William with Maria	-0.97
William with Suzan	-0.69
William with William	1

For the Predict and Recommendation Computation, we will choose the Weighted Sum of Other's Ratings approach. We calculate the results for K values, where K is the size of the neighbors set.

K	Predication
1	6
2	6.5
3	5

So probably William will like Harry Potter.

4.3 Item-Item CF

4.3.1 Assumption

If a user X likes an item Y, and the item Z is similar to the item Y, then X is likely to like Z.

Item-Item CF appeared because the User-User suffers from sparsity [1] and its algorithms need big computation time. (Even the incremental SVD seems to be slow. Plus, the users' preferences change quickly, and we need to adapt to these changes).

The motivation for building item-item CF instead of user-user CF is listed in section 4.3.2, and the generic algorithm for any item-item CF is described in section 4.3.4

[1] In the case of Amazon for instance, there are more than 3 million items, most of them have a small number of ratings, and each user would rate maximum 100 items. Thus, the matrix is extremely sparse.

4.3.2 Motivation

1. The number of users is, in the most cases, greater than the number of items. As a result, calculating the similarities between items should require fewer calculations than the calculating the similarities between users.
2. The relationship between pairs of items is almost stable comparing to the relationship between pairs of users. That is because the average item has many ratings than the average user. For instance, in an online store that has 10 million customers and 10 thousand items, if the average user has rated 50 items, the average item has been rated by 50 thousand users. As a result, adding (or changing) some ratings would not change the ultimate similarities between items[1]. Thus, we have the opportunity to build a model that saves the similarities and use it late. This is not available in User-User because adding (or changing) 2 or 3 ratings for a user would shift its neighborhood. That supports the idea that it is better to depend on the similarities between items rather than the similarities between users.
3. Truncate model. We don't need to save the similarities between an item and all the other items in the system, but we can truncate the model to save just a specific number of items. This number is called the *Model Size*. This can't be done accurately for the users because users might agree on some topics and not agree on another topics. Plus, Users' ratings keep changing. For instance, if a user watches a movie today, he might give it 5 stars, while if he wanted to rate it after 1 year, he might not give it 5 starts because he would have watched more movies or not interested in it as he was the first time he watched it. Nevertheless, we could truncate the model in a User-User CF, but that doesn't mean an accurate results comparing to Item-Item CF approach.

4.3.3 Characteristics

All Item-Item CF systems share the following characteristics:

- Ratings: they all provide a way to allow users to rate the items
- Model Builder: they all provide a way to build a model, as will be described in section 4.3.5

[1] In other words, we do not have to re compute the similarities on the fly.

- Items' Similarities: they all calculate the similarities between items using different methods, as will be described in section 4.3.6.
- Personalized Recommendations / Predictions: they all provide recommendations and predictions depending on item score aggregation functions, as will be described in section 4.3.7.

4.3.4 Generic Algorithm

1. *Item Similarities Computation*: pre compute the model that contains the similarities between all pairs of items. as described in section 4.3.5.
2. *Predication Computing*: predict User-Item ratings, as described in section 4.3.7

4.3.5 Building Model Algorithm

The generic approach for building the model is [5]:

$$for\ j\ \rightarrow 1\ to\ n$$

$$do \begin{cases} for\ i\ \rightarrow 1\ to\ n \\ do \begin{cases} if\ i\ \neq j \\ then\ \mathcal{M}_{i,j} \rightarrow sim(R_{*,j}\ , R_{*,i}) \\ else\ \mathcal{M}_{i,j} \rightarrow 0 \end{cases} \\ for\ i \rightarrow t\ o\ n \\ do \begin{cases} if\ \mathcal{M}_{i,j}\ \neq among\ the\ k\ largest\ values\ in\ M_{*,j} \\ then\ \mathcal{M}_{i,j} \rightarrow 0 \end{cases} \end{cases}$$

$$return\ (\mathcal{M})$$

Where M is the model (Table 4-3 is an example). $R_{*,j}$ is the ratings of all the users for the item j. R is the matrix that contains the user's rating, in which users are columns and items are rows or vice versa. $sim(R_{*,j}\ , R_{*,i})$ is the similarity between the two items i, and j, in which each item is represented as the ratings of all the users for it. The complexity of building the model is $O(|I|)^2$ because we need to compute the similarity for every pair of items. (In the calculating the similarity section 4.2.5, if we use a symmetric method, we can cut the complexity of building the model to $O\ (I)$).

We call K the *Model Size*, if we select K a too small value, we get inaccurate results because we wouldn't be considering so many neighbor items. In the other hand, if we select a large value for K, we will get too much noise. MovieLens study suggests that 20 works very good almost always [20] while another study suggest that k could be a number in this range: $10\ \leq k\ \leq 30$.

Table 4-3 Item-Item model example

	Item 1	Item 2			Item m
Item 1	0	$sim(1,2)$			$sim(1,m)$
Item 2	$sim(2,1)$	0			$sim(2,m)$
Item m	$sim(m,1)$	$sim(m,2)$			0

4.3.6 Item Similarities Methods

When thinking about similarities between two items, we need first to consider two important things [5], which are *Customer Discriminations*, and *Symmetric Similarities*:

Customer Discriminations cares about the question: should we differentiate between customers that buy a lot of items and those who don't? If we have two customers, one of them has bought 100 items, and the other has bought just 8 items, should both of them participate equally to the similarity between two items? Experiments show that the customers that buy less number of items are more reliable when computing the similarities because the other group tends to buy everything (or a big number of items).

Symmetric Similarities cares about the question: is the similarity between $an\ item\ i$ and $an\ item\ j$ equal to the similarity between $an\ item\ j$ and $and\ item\ i$?

Symmetric similarity → $sim(i,j) = sim(j,i)$

Asymmetric similarity → $sim(i,j) \neq sim(j,i)$

We think about this situation when we have two items that have been purchased in different frequencies. Let's say that 1000 users have purchased the item i, and 5 users have purchased another item j, and suppose that the number of times that both the item i and the item j were purchased together is 5. As a result, the number of times both the item i and the item j were purchased together is so less than the number of times the item i was purchased. Thus, from $i's$ point of view, the similarities is small, but from $j's$ point of view, the similarity is high because whenever someone buys j, she/he buys i as well.

There are many ways to calculate the similarities between two items, we will list four of them.

1. *Cosine-based similarities:*

Each item is represented as a vector of m dimension in the user-space [1]. The similarity between an item i and an item j is the cosine of the angle between their vectors, as illustrated in Equation 6.

Equation 6 Cosine-based Item-Item similarity

$$sim(i,j) = \cos(\vec{i},\vec{j}) = \frac{\vec{i} \cdot \vec{j}}{\|\vec{i}\| * \|\vec{j}\|}$$

Where $\cdot$ is the *Dot Product* of the two vectors. This similarity method is symmetric since the cosine function is symmetric by its natural. Regarding the *Customer Discrimination* consideration, we can solve it by scaling the vectors to their corresponding unit vector. In this way, the customers who have purchased fewer items will contribute a higher weight to the *Dot Product* than the those who have purchased more items.

2. *Correlation-based similarities:*

We should consider just the users who have rated both items, and we will call them U. The similarity is as illustrated in Equation 7

Equation 7 Correlation-based Item-Item similarity

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\sqrt{\sum_{u \in U}(R_{u,i} - \bar{R}_i)^2} \sqrt{\sum_{u \in U}(R_{u,j} - \bar{R}_j)^2}}$$

Where: $\bar{R}_j$ is the average rating of the jth item

3. *Adjusted Cosine Similarities:*

This approach is similar to other two approaches, but it is different than the cosine similarities that it takes into consideration the scale of each user's ratings that because there are users always rates high and there are users always rates low.

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \bar{R}_i)^2} \sqrt{\sum_{u \in U}(R_{u,j} - \bar{R}_u)^2}}$$

Where: $\bar{R}_u$ is the average of the uth user's ratings.

[1] Here each item is represented by all the users, not just the users who have rated both items

4. Conditional Probability-Based Similarities:

The idea here is the similarity between an item i and another item j is the conditional probability of purchasing the item i knowing that the user has already purchased the item j

$$P(i|j) = \frac{Freq(ij)}{Freq(j)}$$

Where $Freq\ (ij)$ is the number of users that have purchased both the item i and the item j. This similarity measure is asymmetric because $P(i|j) \neq P\ (j|i)$

The asymmetric measure will lead to wrong similarities when the items have been purchased in big difference frequencies. Two proposed solutions are to divide the probability by a number connected to the frequently [30]

$$sim(i,j) = \frac{Freq(ij)}{Freq(i)\times\ (Freq(j))^{\alpha}}$$

Where α could have values between 0 and 1.

$$sim(i,j) = \frac{\sum_{\forall q:\, R_{q,j} > 0} R_{q,j}}{Freq(i)\times\ (Freq(j))^{\alpha}}$$

4.3.7 Predication Computation

In this stage, we already have the model ready. Thus, we know the most similar items to each item. The system extracts I_u, which is the list of items that the active user has rated. Then the system calculates the similarity $sim(i,j)$ between each item $i \in I_u$ and $j \in I$. The system then uses one of the Predication Methods to calculate the predications or Recommendations. We will list two predication methods, which are *Weighted Sum* and *Regression*.

The user's ratings could be represented as a vector of m items, which each value is the rating of that user to that item.

Table 4-4 The ratings of the active user in item-item CB

$r_{u,1}$	$r_{u,2}$	$r_{u,3}$					$r_{u,m}$

1. Weighted Sum:

In order to find out the prediction of the user u_a against an item i. We check the most similar items to the item i

Table 4-5 The similar items to the item i

$sim(i,1)$	$sim(i,2)$	$sim(i,3)$					$sim(i,m)$

Here the system computes the sum of ratings giving by the user u_a on the items similar to the item i where each rating is weighted by the similarities between those similar items and the item i

$$P_{u,i} = \frac{\sum_{all\ similar\ items,N}\left(S_{i,N}\ *\ R_{u,N}\right)}{\sum_{all\ similar\ items,N}\left(\left|S_{i,N}\right|\right)}$$

2. Regression:

Here we don't use the rating $r_{u,i}$ of a user u_a against the item i, but instead we use $r'_{u,i}$ based on a liner regression model where:

$r'_{u,i} = \ \alpha\, r_{u,i} + \beta + \varepsilon$

Where there are many ways to calculate $\alpha, \beta, and\ \varepsilon$

There are other equations for regression models. [31]

4.4 Summary

4.4.1 CF advantages

- It is simple to build because it just depends on ratings
- It gives novel recommended items because it depends on other people opinion.
- Its performance is better than CB recommender especially item-item approach because of the model pre calculation feature.

4.4.2 CF disadvantages

- If the system does not have many ratings for a user, it can't generate recommendations for her/him.
- It does not take the content of the items into considerations so its quality is not as good as CB.

“The nitrogen in our DNA, the calcium in our teeth, the iron in our blood, the carbon in our apple pies were made in the interiors of collapsing stars. We are made of stars’ stuff.”

Carl Sagan

5 Recommender Systems’ Evaluation

The aims of this chapter are four-fold:

- First, provide the reasons why evaluation is important.
- Second, introduce accuracy metrics for RS.
- Third, introduce decision support metrics for RS.
- Fourth, introduce ranking metrics for RS.

5.1 Introduction

We need to evaluate any RS because there are so many of algorithms, so we need to evaluate them in order to know which one to pick. Also, we have noticed from commercial experiences lessons that the things that researchers love to measure actually don't matter at all in the business area. A banana RS, listed in the section 5.1.2 is a good example. Moreover, sometimes using a specific measure is good in some context, but not good in another one. For example, if MAE, which is an accuracy measure that will be described in section 5.2.1, gives 70% correct recommendations in a movie context, that doesn't mean it will perform the same in a music context. In general being accurate is not enough [32].

The goal of evaluation is to understand and evaluate both the goodness and the quality of the recommendations, and the quality of the recommendation's algorithms

When we evaluate any RS, we should think of three main criteria (themes), as described in section 5.1.1. There are many evaluation families such as Accuracy metrics family, described in section 5.2, decision support metrics, described in section 5.3, Ranking family, described in section 5.4, and other metrics described in section 5.5.

5.1.1 Themes

The three themes that we should think about when it comes to evaluation are:

1. Predication verse Top N

Evaluating predications is not the same as evaluating the Top-N recommendations list. Evaluating predications is mostly about accuracy (Example: is the predication that the RS generated correct), but it is somehow also about decision support metrics (Example: we will recommend items that their predications are more than 3.5 stars[1] but what if the user dislikes it). On the other hand, evaluating Top N is more about ranking because the list comes in order and the order is very important[2]. It is also somehow about decision support metrics by: are the items the system has picked good comparing to those items the system could have picked.

1 Whenever we use starts for predications, we mean 5 stars rating schema.
2 The best place to hide a criminal is the second page of Google's search results.

2. Unary Data

Many measure are designed to work with scale rating system such as 1 to 5
Some measures don't work with unary data.

3. Dead Recommendation versa Live Recommendation

Retrospective (dead) evaluation approaches tries to see how the RS would have predicated items that have already been rated or purchased. Here we can't know if the system does a good job at predicting novel things because the dead data is not novel data; it is what the users have already received and consumed.

Prospective (live) evaluation approaches tries to see how the recommendations are being consumed, either explicitly by asking the users or implicitly by monitoring their actions toward the items. The live evaluation is hard because the explicit way may distract (upset) users, and the implicit way may not be accurate.

5.1.2 Commercial Look

Nobody in the business area cares actually about accuracy, but what impressed them is increasing the sales. From the business point of view: a good RS is the one that leads to make more money. Interestingly, generating correct recommendations doesn't necessary increase the sales. For instance, banana is one of the most items that people buy in USA. If we build a RS that predicts *people will buy banana*, that won't be a good achievement at all, because they already buy Banana, the RS doesn't change their behavior at all, it just predicts what they were going to do anyway. The RS has to bring some novelty to the business.

Businessmen use specific measures. Such as: Lift, cross-sales, up-sales, and conversions; they all measure how much more money do they make, or how many additional sales do they make.

If users will look for just the first five recommendations, why do we care about the other recommendations?

5.2 Accuracy Metrics

These set of metrics measure how far a giving prediction is from the underling truth data of the user's ratings. In other words, they measure how good a recommender is at predicting the rating a user would give to an item. So, it is predications verses Ratings [33]. If a user has rated 100 items, the system covers one item and tries to predict it and then checks the difference between the predicated value and the rated one. We will list three of the accuracy metrics family, which are MAE, MSE, and RMSE.

5.2.1 Mean Absolute Error (MAE)

MAE is calculated as illustrated in Equation 8. The error is the difference between the predication and the actual rating ($P - R$). The absolute is the absolute value of the error ($|P - R|$), and the mean is the average.

Equation 8 MAE equation

$$MAE = \frac{\sum_{ratings} |P - R|}{\# \, ratings}$$

5.2.2 Mean Square Error (MSE)

MSE is calculated as illustrated in Equation 9 [33]. There is no need to do the absolute part as in MAE. This matric penalizes large error more than small.

Equation 9 MSE equation

$$MSE = \frac{\sum_{ratings} (P - R)^2}{\# \, ratings}$$

5.2.3 Root Mean Square Error (RMSE)

RMSE is calculated as illustrated in Equation 10.

Equation 10 RMSE equation

$$RMSE = \sqrt{\frac{\sum_{ratings} (P - R)^2}{\# \, ratings}}$$

5.3 Decision Support Metrics

They are designed to try to give a clue at how well the recommender does at helping people distinguish between good things and bad things. Any RS should help users to make good decisions [1]. The Decision Support Metrics measure the goodness of the system from that point of view. The philosophy of decision support metrics is different than the one of accuracy metrics. In accuracy metrics if we predicate an item as 3.5 stars, but it is actual rating is 5 stars, that is a big mistake, but it might not be a problem for decision support metrics because the system will recommend all the items that their predications is equal or

[1] A "good decisions" could have many interpretations. For instance: buying the item is a good decision. Plus rating the recommendations high is a good decision.

greater than 3.5 stars. The same thing for a predication 2.5 stars where the actual rating is 1 star, the system won't recommend that item anyway so from a decision support point of view, the RS is doing good but from an accuracy point of view, the RS is not doing that good.

We need to differentiate between *Errors* and *Reversals*.

Errors are wrong predications and wrong recommendations. An example of wrong predications is when a good song from the user's point of view receives bad prediction. An example of wrong recommendations is when a bad book from the user's point of view appears in the top N recommendations list.

Reversals are large mistakes; typically reported as more than a certain points off on a scale. For instance, a movie that the user would rate 4 was predicated by the system as 1. The intuition here is that if the recommender is that far, it will lead the users to lose confidence on the whole system. The philosophy of reversals is that usually the users are sensitive when the recommendations are too wrong, they would lose trust in the system.

Three of the decision support metrics family, which are Precision and Recall, F-Measure, and MAP, will be listed:

5.3.1 Precision and Recall

Precision is the percentage of the good recommendations [1] generated comparing to the whole recommendations generated [34]. It is always a rate between zero and one, in which zero means all the recommendations are irrelevant (not good), and one means all the good recommendations that the system's has were generated (retrieved). In other words, the goal of precision is making sure what gets recommended is useful (good) from the user's point of view. High precision filter makes sure that the user doesn't waste time looking for recommendations that are not good (not useful, not relevant to his interests). The assumption is that the system has too many stuff that the user doesn't want, but we are trying to recommend stuff that the user really wants (likes). Equation 11 shows how to calculate it.

Equation 11 Precision equation

$$P = \frac{number\ of\ good\ recommendations\ generated}{number\ of\ recommendations}$$

Recall is the percentage of the good recommendations generated comparing to the whole good recommendations that the system's database has. It is always a rate between zero and one, in which one means all the good (useful) recommendations, from the user's point of

[1] A good recommendation is the one that the user would rate highly, or the one that the user makes actions against it. Such as purchasing, or clicking like

view, were recommended (generated), and zero means none of the generated recommendations are good (useful). In other words, the goal of recall is not missing useful stuff. Equation 12 shows how to calculate it.

Equation 12 Recall equation

$$R = \frac{number\ of\ good\ recommendations\ generated}{number\ of\ good\ recommendations\ the\ system's\ database\ has}$$

Problems with Precision and Recall

1. It needs a ground truth for all items: we need to know already, and for all items, if they are good or not for the all users. We don't have a ground truth and if we did we wouldn't need a recommender system. Creating a test dataset out from already known ratings usually solve the problem. However that leads to fake results because this workaround doesn't measure the accuracy among the whole system continuously, but it measures the accuracy amongst a selected dataset, which is restricted for time and users. Jester [35] is a joke recommender system, which follows the user-user Collaborative Filtering approach and that uses *Eigentaste* algorithm [36], address this problem by letting the users rate the jokes immediately. Though the rating idea exists in all recommender systems, but the natural of Jester [1] makes it easy for users to rate the jokes immediately[2]. As a result, they could argue that they have a ground of truth.
2. It's targeted against the whole dataset [3]: It's rarely that we care about the whole set of recommendations generated by the recommender. Often we care about the top N recommendations. Researchers suggest modifying the precision to *Precision At N* measure [34], which is the percentage of the good generated items in the top N recommendations generated by the recommender, as illustrated in Equation 13.

[1] In other domains such as movies or book, users can't give immediate feedback on the recommended items. That's where Jester exceeds; because users can rate the joke right away.
[2] The user reads a joke, and she/he can rate the joke at the same moment. While in the books context or movie contexts, the user needs to watch a movie or read a book in order to rate it, if she/he comes back to the system and does that.
[3] Though that's an advantage point in the information retrieval area, it's not in the field of recommendation systems.

Equation 13 Precision at N equation

3. $$P@n = \frac{number\ of\ good\ recommendations\ generated\ in\ the\ top\ N\ recommendation}{N}$$ [1]

5.3.2 F-Measure

It is the metric that comes from combining recall and precision [34] .

$$F1 = \frac{2PR}{P + R}$$

5.3.3 Mean Average Precision (MAP)

It is a precision-recall related metric. In the information retrieval field, MAP for a set of queries is the mean of the average precision scores for each query [34]

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$$

Where:

$$AveP = \frac{\sum_{k=1}^{n} P(k) * rel\ (k)}{number\ of\ retrievd\ documents}$$

And Q is the number of queries.

Researchers tried to adopt that metric to the recommendation systems field of study by considering users as queries.

5.4 Rank Metrics

They are designed to tell how good the ranking of the recommendation is. In other words, how bad stuff the user will face before having the first good recommendation from his/her point of view. The philosophy of these metrics is that users will look at just some of the recommended items, not the whole list, and usually those items are the ones at the top of the recommendations list.

5.4.1 Mean Reciprocal Rank (MRR)

How bad items the user will face before getting the first good item. For example, suppose we have RS for books. Table -5-1 represents the entry data, and the recommendations, where the first column simulates users preferences [2], and the second column contains a list of the books the RS recommends. The Reciprocal Rank column is calculated according to Equation 14.

[1] There is no R@n, *Recall At N*, measure because it'd be equal to P@n

[2] Such as books the user has likes, or artists the user is interested in … etc.

Equation 14 Reciprocal Rank equation

$$Reciprocal\ Rank\ = \frac{1}{i} : i\ is\ the\ order\ of\ the\ first\ good\ item$$

Table -5-1 Sample data for MRR example

User's Preferences	Recommendations	First Correct Recommendation	Rank	Reciprocal Rank
Stephen Hawking	A Brief History of Time, Last Three Minutes, Quantum Gravity	A break History of Time	1	1/1
Paul Davies	1984, Billions and Billions, Last Three Minutes	Last Three Minutes	3	1/3
Animal Farm	Hamlet, 1984, The Lord of the Rings	1984	2	1/2

Equation 15 Mean Reciprocal Rank equation

The equation to calculate the Mean Reciprocal Rank is illustrated in

Table 5-2 MRR equation

$$MRR = \frac{1}{|N|} \sum_{i=1}^{|N|} \frac{1}{rank_i}$$

Where N is the number of times we try the RS. In the example above:

$$MRR = \frac{1}{3}\left(\frac{1}{1} + \frac{1}{3} + \frac{1}{2}\right) = \frac{11}{18} \approx 60\%$$

To apply MRR, we assume that:

- There is no list of recommendations that doesn't have a good result
- There aren't two results with the same rank

There is a study tried to evaluate a CF RS using MRR [37]

5.4.2 Spearman Rank Correlation

It is the Pearson correlation of the ranks of the recommendations [29]. Thus, we need to have something to compare with. So, we assume that we have already a ground truth ranking. In other words, this metric tries to evaluate how good the rankings are comparing to the actual rankings[1].

Equation 16 Spearman Rank Correlation equation

$$SRC = \frac{\sum(i - \mu_i)\left(j - \mu_j\right)}{\sqrt{\sum(i - \mu_i)^2}\sqrt{\sum\left(j - \mu_j\right)^2}}$$

Where i is the ranking from RS, j is the correct ranking, and μ_i is the average of is

There is no problem if the correct ranking list has two items of the same ranking value.

Problem

The problem of SRC is that it penalizes inaccuracy in the tail of the recommendation list, which the user will likely never see, to the same degree as inaccuracy in the top-predicted items [29]. In other words, it handles equally both situations where an item is ranked 110th by RS, while its actual "correct" ranking is 115th and the case where the RS ranking is 1st while the actual ranking is 5th. Thus, we would prefer an algorithm that weights things at the top of the things more heavily. DCG, described in the next section solves that problem.

5.4.3 Discounted Cumulative Game (DCG)

It measures the utility of an item at each position of the recommendations list. It is being calculated as illustrated in Equation 17

Equation 17 DCG equation

$$DCG(L) = \sum_{i}^{all\ recommended\ items\ in\ the\ List\ (L)} u(i) \times d(i)$$

Where L is the recommended List, $u(i)$ is the utility of the recommended item (i), $d(i)$ is the discount of the recommended item (i). However, we need to compare the DCG of the recommendations list L, with the DCG of the perfect ranking list. Equation 18 does that.

Equation 18 Normalized DCG equation

$$nDCG(L) = \frac{DCG(L)}{DCG(L_{Perfect})}$$

There are many proposals to calculate the utility $u(i)$, such as: $u(i) = r_{a,i}$ where $r_{a,i}$ is the rating of the user a to the item i, and another way could be :

[1] The ones that comes from the ground truth rankings

$$u(i) = \begin{cases} 1 \; if \; the \; user \; clicks \; on \; the \; item \\ 0 \; otherwise \end{cases}$$

In generate, the $u(i)$ should reflect how much the user finds that the item i is valuate (or invaluable).

The discount $d(i)$ should be a function that decrease when we go down in the recommendations list so that the items in the top of the list are more important. There are many proposals to calculate the discount $d(i)$, such as:

$$d(i) = \frac{1}{\min{(1, log_2 i)}}$$

5.4.4 Fraction of Concordant Pairs (FCP)

This matric cares about how many pair of items in the recommendations list are at the same order as the actual order that the user rated.

$$FCP = \frac{N}{Number \; of \; pairs}$$

Where N is the number of pairs that appeared in the same order in both the recommendations list and the list obtained by users rating. For example, suppose we have a music RS that produces a list of recommendations as listed in Table 5-3

Table 5-3 Sample data for FCP matric

Recommendations	Ranking by RS	Ratings by William	Perfect Ranking
Hello (Lionel Richie)	1	4	3.5
I'll always love you (Whitney Houston)	2	3	5
I just called to say I love you (Stevie Wonder)	3	4	3.5
Delilah (Tom Jones)	4	5	1.5
Sole Mio (Pavarotti)	5	5	1.5

$$FCP = \frac{3}{10}$$

Where 10 is the number of pairs, and 3 is the number of the following set:

{(Hello, I'll always love you), (Hello, I just called to say I love you), (Delilah, Sole Mio)}

On the same data set, and assuming that the "good item" is the one rated 4.5 and more, the Reciprocal Rank is:

$$RR = \frac{1}{4}$$

5.5 More Metrics

This set of metrics is closely related to business goals and user experience. It is not always about making good recommendations, or good predications. As the banana example, illustrated in section 5.1.2, shows that sometimes business owners don't care about accuracy, but they rather care about making more money and more sales.

The current evaluation mechanism may be good from systematic point of view, but if we look at Amazon RS for books, it will recommend many books from the same category in the top N list, and that is, in some cases, considered wrong. For example, it is less likely that a user wants to buy a second TOEFL book after buying the first one. Moreover, diversity in recommendations gives the user a trust on the RS. This image is way much important than the number in any evaluation system. That is why many evaluation metrics were developed such as Coverage, Diversity, User retention, and Recommendation uptake.

5.5.1 Coverage

It is one of the oldest measures used. It is the percentage of the products that the data source has and which the RS can make prediction about. In other words, it measures the number of items the RS has enough data on to actually recommend them to the users.

5.5.2 Diversity

It measures how different the items that are recommended. It doesn't make sense in a predication context, but it only makes sense in a top N context.

5.5.3 User retention

Do people stay using the recommender system after 12 months?

5.5.4 Recommendation uptake

Do people listen to the music that the system suggests? Do they watch the movies?

6 The Semantic Web Technologies

إنّي أحبُّ وبعضُ الحبِّ ذبّاحُ	هذي دمشقُ وهذي الكأسُ والرّاحُ
لسالَ منهُ عناقيدُ وتفّاحُ	أنا الدمشقيُّ لو شرّحتمُ جسدي
سمعتمُ في دمي أصواتَ من راحوا	و لو فتحتُم شراييني بمديتكم
وما لقلبي –إذا أحببتُ- جرّاحُ	زراعةُ القلبِ تشفي بعضَ من عشقوا
فالنهد مستنفر و الكحل صبّاح	الا تزال بخير دار فاطمة
فهل عيون نساء الشام أقداح	ان النبيذ هنا نار معطرة
و للمآذنِ كالأشجارِ أرواحُ	مآذنُ الشّامِ تبكي إذ تعانقني
وقطّةُ البيتِ تغفو حيثُ ترتاحُ	للياسمينِ حقولٌ في منازلنا
فكيفَ أنسى؟ وعطرُ الهيلِ فوّاحُ	طاحونةُ البنِّ جزءٌ من طفولتنا
ووجهُ "فائزةٍ" حلوٌ و لماحُ	هذا مكانُ "أبي المعتزِّ" منتظرٌ
فكيفَ أوضحُ؟ هل في العشقِ إيضاحُ؟	هنا جذوري هنا قلبي هنا لغتي
حتّى أغازلها والشعرُ مفتاحُ	كم من دمشقيةٍ باعت أساورَها
فهل تسامحُ هيفاءُ ووضّاحُ؟	أتيتُ يا شجرَ الصفصافِ معتذراً
فوقَ المحيطِ وما في الأفقِ مصباحُ	خمسونَ عاماً وأجزائي مبعثرةٌ
وطاردتني شياطينٌ وأشباحُ	تقاذفتني بحارٌ لا ضفافَ لها
حتى يفتّحَ نوّارٌ وقدّاحُ	أقاتلُ القبحَ في شعري وفي أدبي
أليسَ في كتبِ التاريخِ أفراحُ؟	ما للعروبةِ تبدو مثلَ أرملةٍ؟
إذا تولاهُ نصّابٌ ومدّاحُ؟	والشعرُ ماذا سيبقى من أصالتهِ؟
وكلُّ ثانيةٍ يأتيك سفّاحُ؟	وكيفَ نكتبُ والأقفالُ في فمنا؟
ماذا من الشعرِ يبقى حينَ يرتاحُ؟	حملت شعري على ظهري فأتعبني

The aims of this chapter are three-fold:

1. Describe the history of hypermedia applications
2. Describe The Semantic Web and Its technologies
3. Identify the existing semantic recommender.

6.1 Introduction

To fully understand what The Semantic Web adds to World Wide Web, we should start from the beginning. It is hypermedia and linear media.

6.1.1 Linear media

Human minds work in an associative way, in which we can develop sophisticated knowledge by integrating pieces of information together. The remembering process is associative as well, because a thought triggers another thoughts, which could be linked with a concept that is linked with an idea.

The writing process through linear media is a linearization of the complex non-linearized knowledge stored in author's mind, while the reading process from linear media is a de-linearization of that knowledge [38]. The problem with linear media is that the author is not able to express his ideas in the same way they are arranged in his mind, and it is not necessary for the reader to acquire the knowledge the same way the author intended to. This could lead to wrong interpretation as illustrated in Figure 6-1, where circles represent concepts and ideas in human mind and lines show how they are related. Linearization and De-linearization do not necessary end up with the same concepts in reads' mind linked the same as in writers' minds. Hypermedia tries to solve these problems.

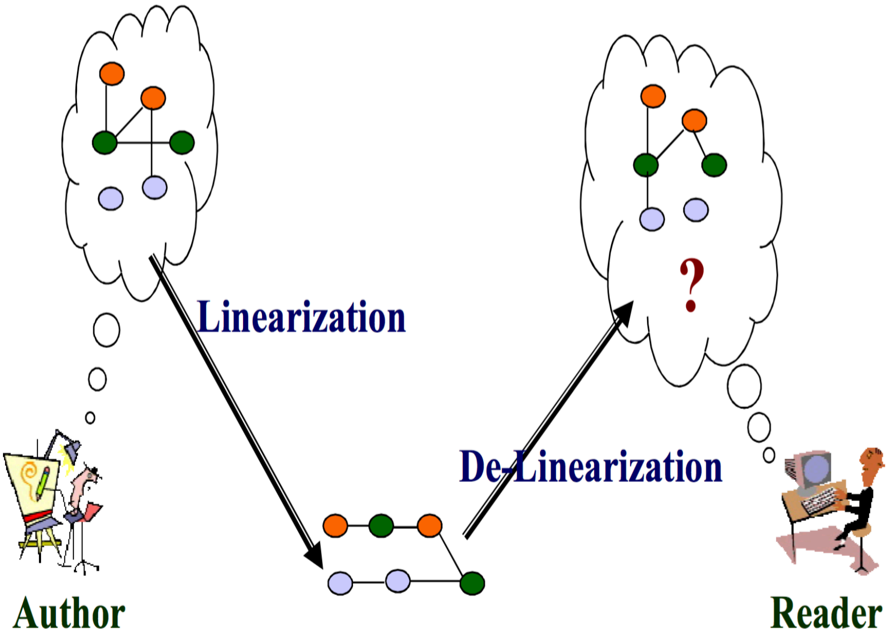


Figure 6-1 an illustration of the linear media's problem
Figure is taking from the lectures of Dr. Ammar Khierbek, Damascus University

6.1.2 Hypermedia

Hypermedia tries to overcome the obstacles of linear media by enabling the writers to compose non-linear structure for their information.

"Schneiderman 1989" defined hypermedia as "A database that has active cross-references and allows the reader to 'jump' to other parts of the database as desired"

"Ted Nelson 1967" defined hypermedia as "a combination of natural language text with the computer's capacity for branching, or dynamic display" [39]

Hypermedia is an application, which uses associative relationships among information contained within multiple media data for the purpose of facilitating access to, and manipulation of, the information encapsulated by the data [39]

Hypermedia is nonlinear. Figure 6-2 and Figure 6-3 show the difference of information linking between linear media and hypermedia respectively.

Figure 6-2 linear linking of linear (traditional) media

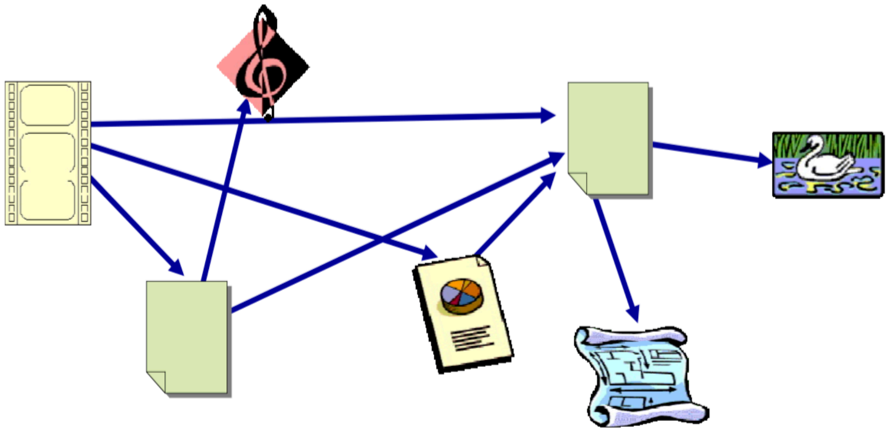


Figure 6-3 non-linear linking of hypermedia
Figure is taking from the lectures of Dr. Ammar Khierbek, Damascus University

The basics concepts in hypermedia are nodes and links.

- *Nodes* are units of information; they are information objects displayed simultaneously in one window or in separated windows.
- *Links* are the way of connecting nodes; they are unidirectional or bidirectional embedded in the information object or dynamically generated.

Hypermedia applications cover almost all the domains. For example, in dictionaries, they were used for lexicographical purposes. In Instruction and Educational Systems, the best example is eLearning applications. In Encyclopedias and Virtual Libraries, they are used for knowledge linking. During the evolution of hypermedia applications, we reached the World Wide Web (WWW).

6.1.3 World Wide Web

Simply, the World Wide Web is a hypermedia application. It was invented by Tim Berners-Lee invented in 1989. Before this date, we had documents that had references to other documents, and these references were in the form of citations. As consumers for one of these documents, we would have to read these citations and send a request, which could be in a mail or another way, to get them. Thus, from the point view of a consumer, as illustrated in Figure 6-4, we had too many documents; some are accessible while others are not.

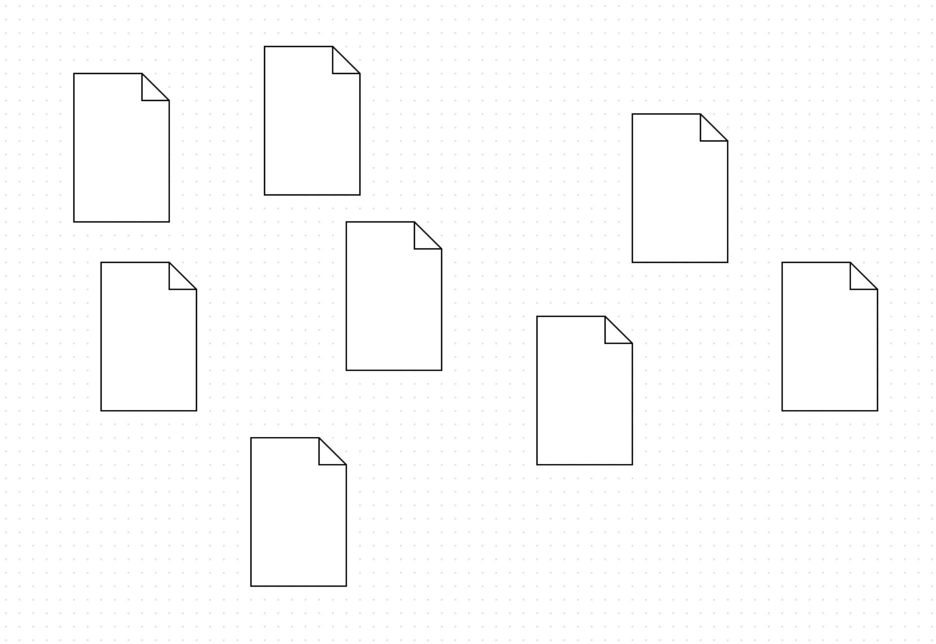


Figure 6-4 the documents before the Web

The Web brings the concept of hyperlink, which is a reference to another piece of information that could be at the same document, or could be another new document. In other way, the documents now, from the point view of a consumer, is linked together, that is why they called this version of the Web as *Web of Documents*, or *Web 1.0*, as illustrated in Figure 6-5

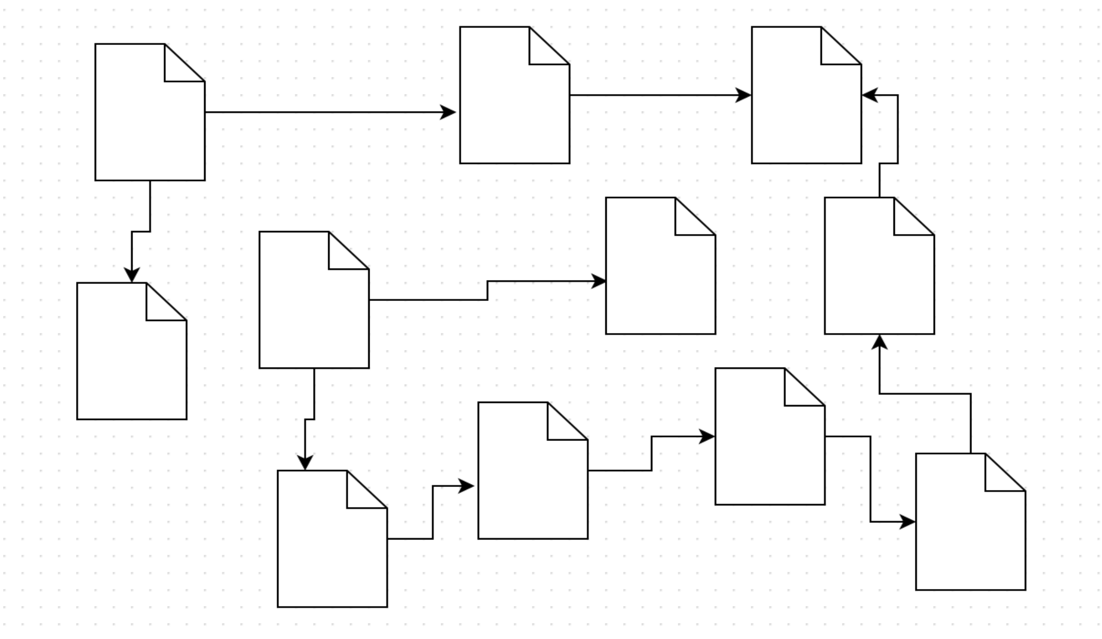


Figure 6-5 Web 1.0, Web of Documents

The main scenario in Web 1 was: authors publish documents, and readers read them. As time goes, new technologies appeared that allowed users to easily publish their content and to interact with each other. They called this kind of Web as *Social Web* or *Web 2.0*.

6.1.4 Web 2.0

It is a new development of *Web 1.0* where users are given the abilities to collaborate and generate content. The HTML pages are not static anymore; they became dynamic. The UGS (user-generated content) leads to the tremendous amount of content that we can find in wikis, blogs, social networks and web applications[1]. We started to have so many web applications, such as Facebook, LinkedIn, Twitter, Gmail, and YouTube. However, the big drawback is that these web applications do not interoperate; they are not integrated together, their models are different, their data structure various so much. For example, if a user updates her/his profile's info on Facebook, LinkedIn does not automatically know about that, and the user would need to re update it again on LinkedIn. Data is not connected together, and here what *The Semantic Web* is trying to solve.

6.1.5 The Semantic Web

The Semantic web is "a web of data defined and linked in such a way that its meaning is explicitly interpretable by software processes rather than just being implicitly interpretable by humans" [40]

Tim Berners-Lee defined The Semantic Web [41] as "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation".

In The Semantic Web, not just the documents are linked, nor just high level hyperlinks between applications, but the data is connected at the lower level, and that's why Tim Berners-Lee called it *Web of Data*, as illustrated in Figure 6-6. Moreover, the data is *understandable* by machines, where it was just *readable* in Web 2.0 and Web 1.0.

[1] Some would call Web 2.0 as Web of Applications because of the countless number of the web apps.

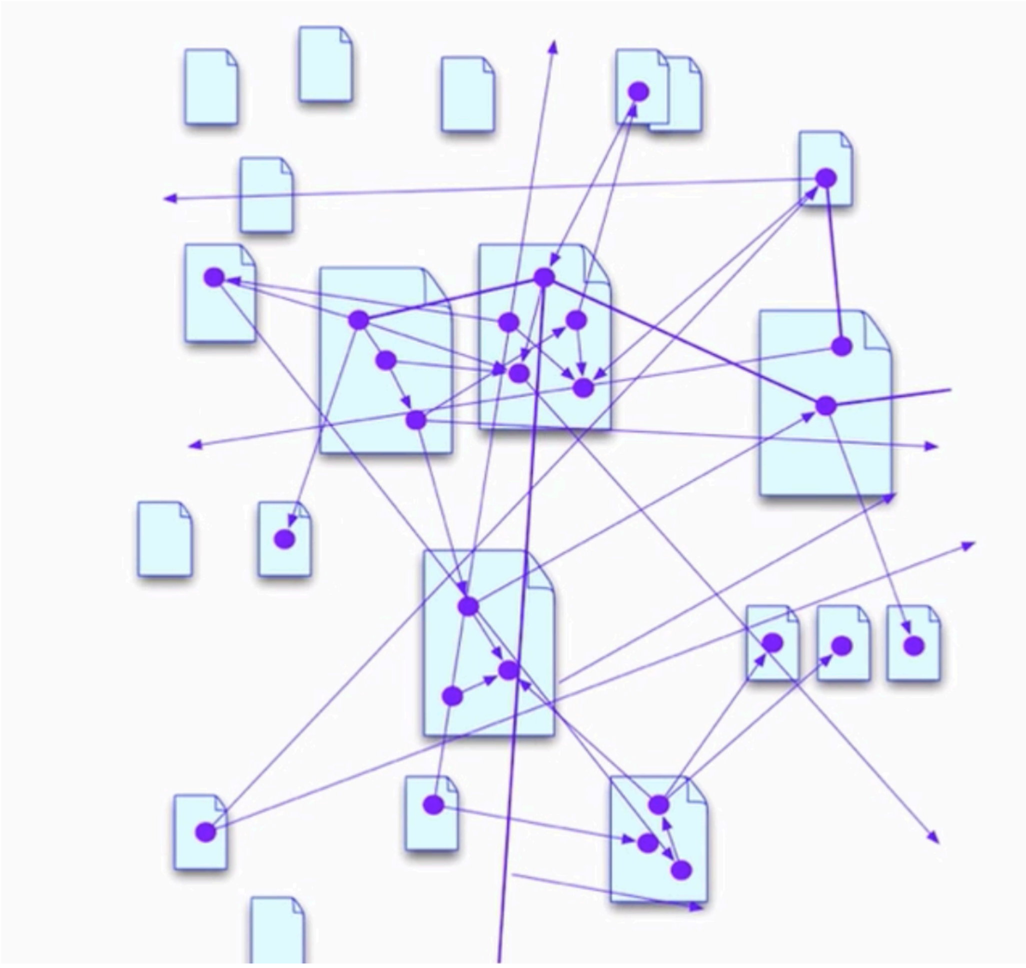


Figure 6-6 the Web of Data

The great advantage is that we do not need anymore to think about a specific document, but we just need to care about data. This is extremely powerful because in the same way Web 1.0 allowed us to not think about where the document is located, The Semantic Web allows us to ignore the document as a whole and just care about the piece of data we are interested in. In other words, The Semantic Web gives us the ability to represent the information at a lower level than documents and yet it is machine understandable. The Semantic Web is based on some concepts, as illustrated in section 6.1.6.

6.1.6 The Semantic Web Concepts

1. Uniform Resource Identifier (URI)

An URI is a compact string of characters for identifying an abstract or physical resource [42]. It is *Resource* because it can be anything that has an identity [42] such as a bed, a graduation project, and a sun glasses. Note that not all the resources are network *retrievable* [42]. Some URIs examples:

- ftp://ftp.sy.is.it/music/classical/dies-iras-requiem-mozart.mp3
- http://www.physics.uio/quantumphysics/gravitational-wave-found-altimately-2016.html

There are many URI schemas that we can use, but Tim Berners-Lee suggests using HTTP URI schema [43] in order to move to Linked Open Data.

2. Ontology

In the informatics context, it refers to the science of describing the kinds of entities in the world and how they are related. In The Semantic Web context, Ontology[1] is a formal specification of a shared conceptualization [44]. The Ontology consists of classes and their relationships, which are modeled using RDF. Thus, in The Semantic Web the schema and the data are modeled in the same way, which allows the machines to query both at the same time. Astrophysics Ontology, for example, could contain the following classes: Scientist, Galaxy, and Meteor. Some relationships between them could be: operatesOnGalaxy that link Meteor to Galaxy, or discoveredBy that link Meteor to Scientist.

6.1.7 The Semantic Web Technologies

In order to achieve the great goal of The Semantic Web, there were inventions for new technologies. Figure 6-7, which is taken from Wikipedia[2], illustrates the stack of The Semantic Web.

[1] The ultimate vision for Tim Berners-Lee is to have one Ontology represents all the human knowledge. As a result, having more than one Ontology is supposed to be just a temporary stage toward the ultimate goal. So, we will use the word Ontology(ies) and not Ontologies to represent more than one Ontology.

[2] https://en.wikipedia.org/wiki/Semantic_Web#/media/File:Semantic_web_stack.svg

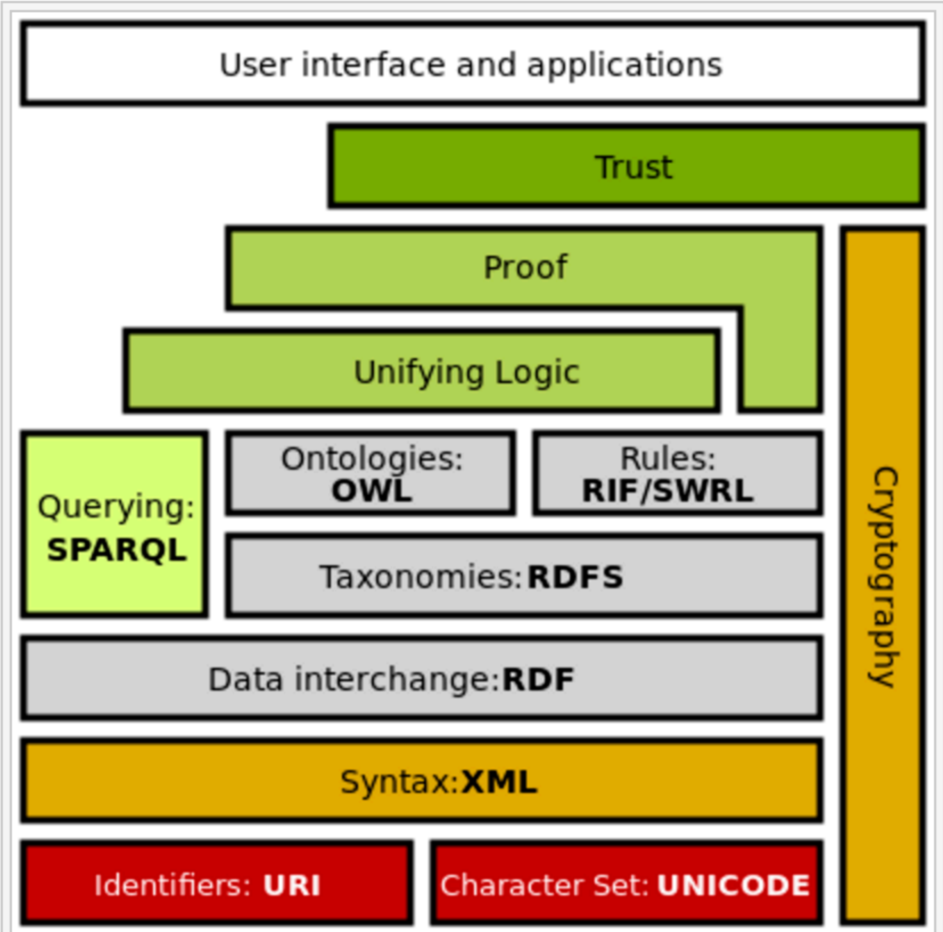


Figure 6-7 The Semantic Web stack

In the scope of work for this thesis, we mostly use RDF (cf. section 1), RDFS (cf. section 2), OWL (cf. section 3) technologies, and SPARQL (cf. section 4).

1. The Resource Description Framework (RDF)

It is the abstract data model in The Semantic Web. It is a language for representing information about resources in the WWW [45]. It can be used to represent metadata about Web resources. RDF's model is a triplet, which are resource, property, and value. The *Resource*, sometimes referred to as *subject*, is any thing that has a URI. The *Property*, sometimes referred to as *predicate*, is relationship between resources and/or atomic values. A property enables us to attach information to resources. The *Value*, sometimes referred to as *object*, can be either a literal or a resource as well. For example: we have this fact: "*Arturo Márquez composed Conga del fuego*", which can be represented using RDF's triple as the following:

The resource is http://de.dbpedia.org/resource/Arturo_Márquez

The property is http://music-actions-vocabulary/composed

The value is http://amazing/music/conga-del-fuego

RDF triples lead to a labeled and directed graph, as illustrated in Figure 6-8. Using roles such as if a user likes a resource and that resource has similar *features* with another resource, then this user will probably like this new resource. These roles could be simply a SPARQL query. So, machines can infer that if a user likes *conga-del-fuego*, she/he probably will like the *Sympony_2_in_D_major* for Carl Czerny (assuming that those two music pieces share the same features), and this can be seem as a kind of semantic recommendations.

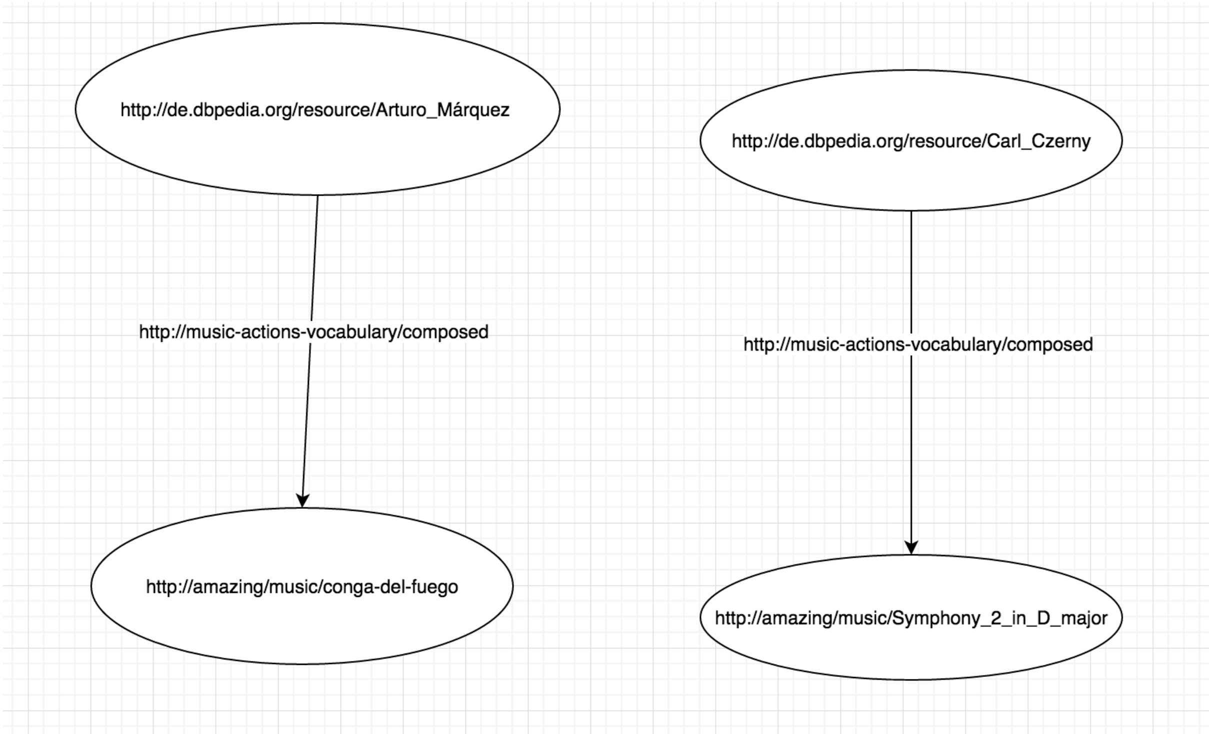


Figure 6-8 RDF graph example

As stated before, RDF is an abstract data model, so we need a way to serialize that model to a way machines are able to read. There are many RDF Serializations, such as RDFa [46], RDF/XML [47, p. 1], Turtle [48], TriG, N-Triples [49], and JSON-LD [50].

2. RDF Schema (RDFs)

RDF Schema provides a data-modeling vocabulary for RDF data. RDF Schema is an extension of the basic RDF vocabulary [45]. For instance, in the previous example about Arturo Marques, we stated that *Arturo Marques* is a resource, but how could the machine understand that? We need to state that explicitly. RDFs provides us with this feature using the class: *rdfs:Resource*. Figure 6-9 completes the definitions in a way machines can understand.

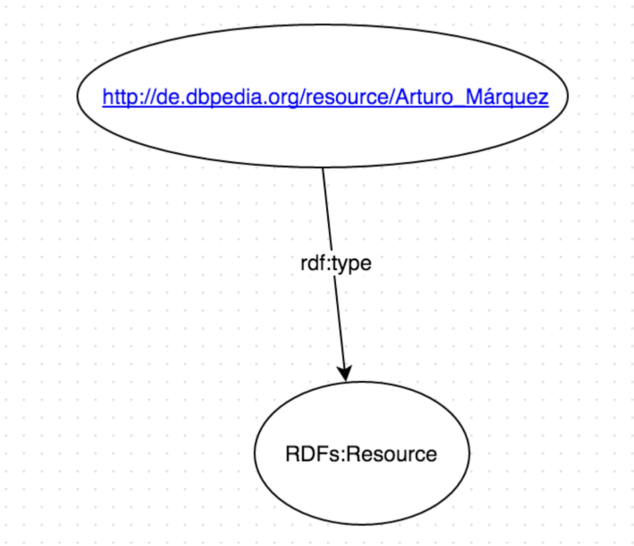


Figure 6-9 Example of using RDFs vocabularies

There are many RDFs resources, with a giving and unchangeable semantic, such as: *rdfs:subClassOf*, *rdfs:domain*, *rdfs:range*, *rdf:predicate* …

RDFs is also used to create/define other vocabularies, such as Friend of a Friend (FOAF). We should always reuse the already-existing vocabularies to make our data linked as most as possible –hint that is the purpose of The Semantic Web. A more expressive vocabulary from RDFs is OWL.

3. *Web Ontology Language (OWL)*

It is an ontology language for the Semantic Web with formally defined meaning [51]. There are many dialects for OWL, each one has different restriction and different expressivity level. It is always a tradeoff between the expressivity level, which is the details level that data can be model to, and the decidability, which is the certainty degree of getting an answer on a query. OWL-Full, for instance, is an OWL dialect, it is very expressive, but it is not decidable. In other words, there is no Description Logic [1] conforms to it, while OWL DL, which is another OWL dialect, is based on SHIQ Description Logic.

OWL has more resources than RDFs.

4. *SPARQL*

It is protocol and language to query the RDF graph. It basically matches the triples in the query with the triples in the RDF store to get the result.

[1] Description logics (DL) is a family of formal knowledge representation languages. (Wikipedia)

6.2 The Semantic Web Recommender Systems

Semantic Recommender is a RS that uses The Semantic Web technologies. Semantic Recommenders are content-based recommenders. Thus, they share the same general algorithm, described previously in section 3.1.2. However, there are no standard methods to represent the items and the users or to calculate the similarities between them. Each Semantic Recommender uses its own way of modeling users and items and of generating the similarities.

6.2.1 Users and Items modeling

Ontology(ies) have been used to model both the items and the users. Instead of thinking of an item as a vector of features (terms), now the items are instances and the features are their predicates and their classes. The same idea is applied for users. [52] builds a Semantic Recommender for tourism in which the point of interests (POI) and the users are mapped to a giving domain Ontology. For item modeling, they mapped the terms of the items (that could have been extracted from POI using TF-IDF if the POI is unstructured text, or could be structured features in a relational database) to their Ontology. Looking at the high level architecture for any CB, illustrated in Figure 3-1, the only new component that we need is a Ontology-based mapper, as illustrated in Figure 6-10.

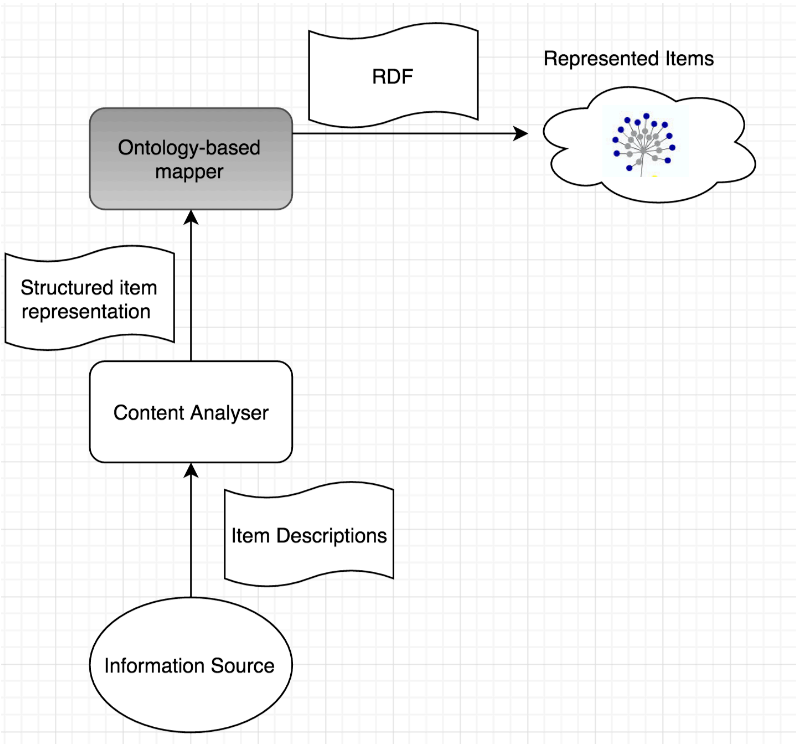


Figure 6-10 CF with Ontology mapper

For user modeling, they would ask the user about his opinion in the first level of classes of the domain ontology. For instance, Figure 6-11 shows the Tourist Ontology, the system would ask the user about his preferences for the first level of class, which are Culture, Nature, Sports and Leisure class. The system assigns two values, which are confidence and preference) for each node in the Ontology, then the system uses the weight of is-a relationship to populate the predication of each class for each user. This approach cares only about is-a relationship between classes (Our proposed solution solves that problem), which leads to the same similarity value for all classes are the same level. (Our proposed solution solves that problem)

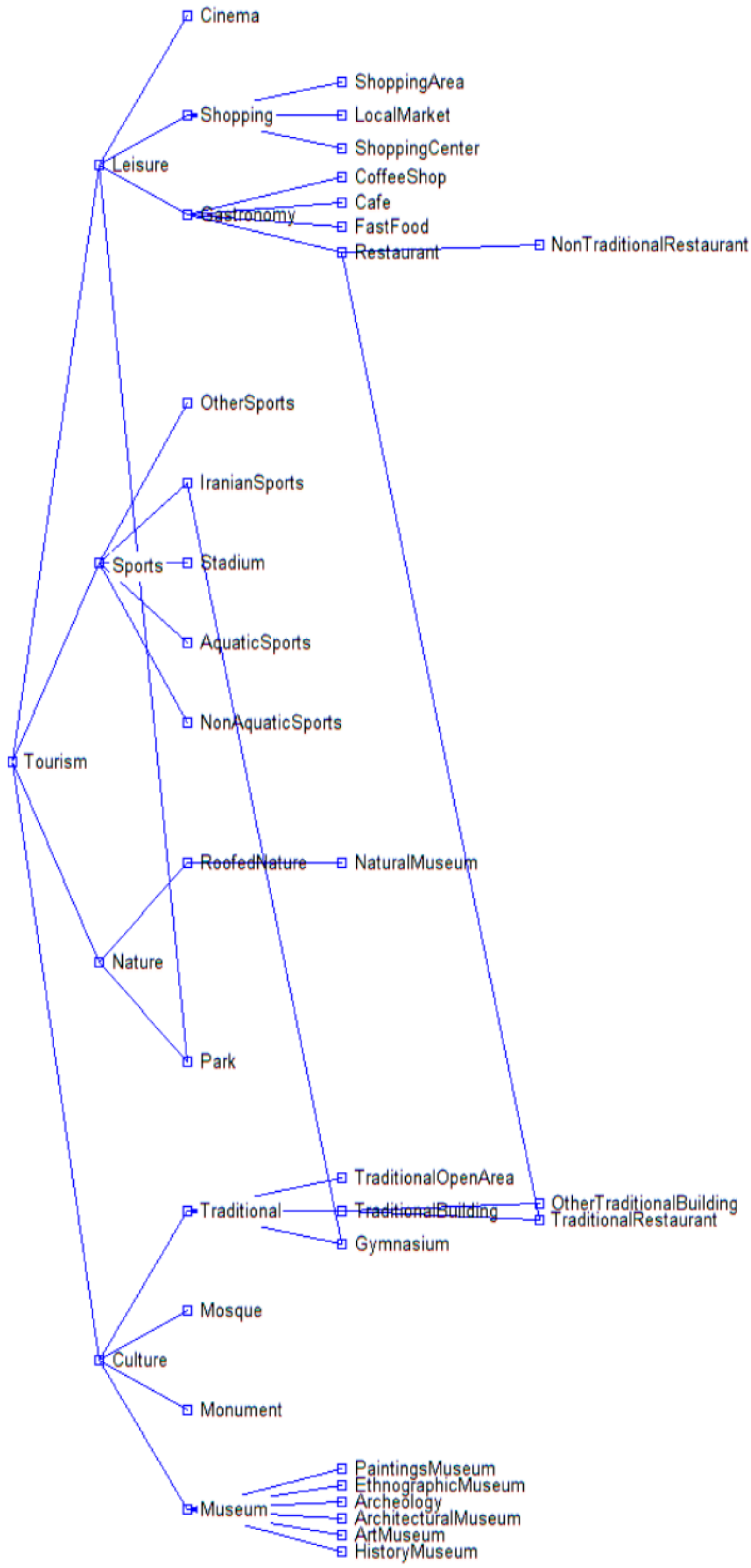


Figure 6-11 Tourism Ontology

Another research [53] used Ontology in a CB system just for user profile classifications. When a user requests a research paper, the system extracts terms for that paper. However, instead of adding the terms directly to the user's profile, the system would use an Ontology to infer more terms (mainly classifications for the extracted terms) and then add those new extracted terms to the user's profile in order generate recommendations using the traditional ways.

6.2.2 Semantic Similarity Approaches

In contrast to traditional CB, in Semantic Recommenders there are no standard equations to calculate the similarities between users and items. However, calculating similarities between two nodes in a RDF graph is possible, and a short introduction to the available methods is illustrated below. They try to find the similarities between two nodes in an Ontology[54] In other words, they try to quantify the relationship between two subjects.

1. *Semantic distance methods*

These methods calculate the similarity between two nodes in a RDF graph as the distance between them. In [55] it was proposed that the distance could simply be the shortage path between them. Another research, [56], considered a weight for each link (for each predicate).

2. *Hierarchical Structure of an Ontology methods*

RDF data is represented as a direct graph, in which each node could have one or more father nodes. The similarity between two nodes can be the distance to the first common ancestor. For example, in Figure 6-12, the distance between *node 3* and *node 1* is 2

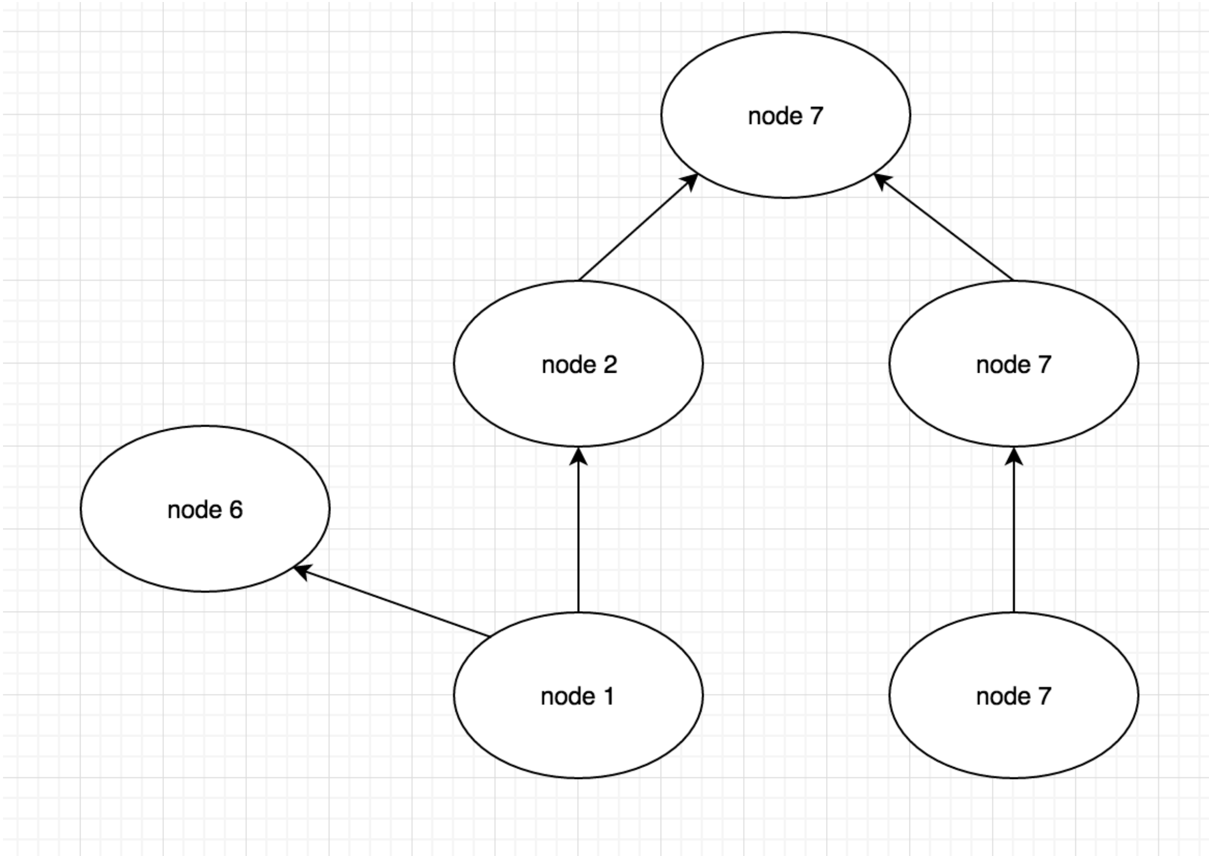


Figure 6-12 Hierarchical Structure of an Ontology example

3. Features similarity methods

Each term[1] has many features[2]. In the case of two terms, we can think of three kinds of features. The first kind is the features that belong to the first term and do not belong to the second one, the second kind is the features that belong to the second term and do not belong to the first one, and the third kind is the features that belong to both terms. [57] argued that the Set theory can be used to achieve a feature similarity function, as described in .

Equation 19 Feature Similarity Method

$$Sim\ (c1, c2) = \frac{|D_1 + D_2|}{|D_1 \cap D_2| + \mu\ |D_1/D_2| + (\mu - 1)|D_2/D_1|}$$

Where $c1$, and $c2$ are the terms the system is trying to find the similarity between, D_1 and, D_2 are the features set for $c1$, and $c2$ consequently. $|\quad|$ is the cardinality.

[1] "Term" in the context of RDF means subject

[2] "Feature" in the context of Ontology-modeled data means predicate

6.2.3 Customize Similarity methods

Some recommenders use customized methods to calculate the similarities between users and items. For instance, a research [58] relied on the Equation 20. The research tries to build a domain-independent RS by using the Ontology domain to improve user's preferences. For each concept in the domain Ontology, the systems assigns for each user two values, which are, *DOI_weight* and *DOI_confidence*. *DOI_weight* represent how much the user likes that concept. They would extract that value either explicitly by asking the user or implicitly by monitoring the user's behavior against items that belong to that concept. The *DOI_confidence* is a predication represents how much the system thinks the user would like that concept.

Equation 20 Customized similarity method equation

$$ItemScore(I,U) = \frac{\sum_{j}^{iConepts(I)} ConceptScore(j,U) * REL(j)}{\sum_{j}^{iConcepts(I)} REL(j)}$$

Where I is an item and U is a user. $iConcepts(I)$ is a set of all the concepts (classes) of the domain Ontology that the item I belongs to.

6.2.4 Summary

Using The Semantic Web Technologies in RS is still immature comparing to traditional way of creating RS. The available attempts are all restricted to a specific domain. The level of customization is still at the lowest possible level, and there is no standard mathematical model to generate the recommendations.

Part II Semantic Recommender

I think; therefore I am

René Descartes

Isn't it a remarkable coincidence almost everyone has the same religion as their parents? And it always just happens to be the right religion. Religions run in families. If we'd been brought up in ancient Greece we would all be worshiping Zeus and Apollo. If we had been born Vikings we would be worshiping Wotan and Thor. How does this come about? Through childhood indoctrination.

Richard Dawkins

7 Proposed Solution

7.1 Introduction

We are building a Semantic Recommender System that falls under Content-based recommender systems category. CB systems proved to give better recommendations than Collaborative-based recommender systems[1] [20]. However, in the current CB approaches, the data is not modeled correctly. Using Ontologies to model the data provides a better modeling quality and thus, more suitable recommendations because better modeled items means more rigorous users preferences modeling abilities which produces more accurate similarities. Moreover, the flexibility of OWL allows machines to infer new knowledge from the already existing data. Plus, working on Ontology(ies), which is one of The Semantic Web techniques that operates closer to the human cognitive model, to generate recommendations offers machines the ability of providing accurate justifications about these recommendations, which is definitely not available in CF approaches, and available in low quality in CB approaches.

Existing CB systems require building a new system for each domain, and existing CF systems require many configurations and adaptation for each domain. On the other hand, the proposed approach works with any domain. As a result, it can generate recommendations regardless of the area it is being applied to.

The proposal is to exploit Ontology-modeled data in order to exploit the intrinsic capability of OWL for data integration and reasoning, and to provide justification. The Value Analysis for the proposal is available on Annex 10.1.

7.2 General Overview

To have a domain-independent RS, we should separate the domain system from the recommender system. The proposed solution consists of the components illustrated in Figure 7-1. The aim of the project is to provide recommendations whatever the domain is. In most cases, companies already have running information systems that present the business to the clients and allow them to buy these products and rate them. They have their data sources that are most likely to be relational databases, but it could be structured or semi structured data sources as well. Those systems and their data sources are presented in the Figure 7-1 as *Domain Business System*, and *Domain Data Source* respectively.

The *Data Modeling* component is responsible for providing a RDF view of the domain data. Two processes are doing that, which are:

[1] As long as there're sufficient information about the users, and sufficient attributes about the items

- *Ontology Creating* process; its responsible of creating the Domain Ontology. This process could be manually in which a domain experts develop an Ontology for the domain business data, or automatic in which the Ontology is being extracted from the data source using many tools such as: D2RQ[1], SPASQL[2], RDQuery[3], DartGrid[4], and P2R[5]
- *Ontology Populating* process; it is responsible of populating the Ontology resulted from the previous process. This process is automated if the data source has a schema. Most of the tools presented in the previous process could be used in the process as well.

The result of the *Data Modeling* component is RDF triples being inserted in the *Domain Triple Store* that forms together with the Recommender Ontology, described in section 7.5 and that is built upon the competency questions listed in section 7.4, the input to the *Configuration* process, which its output is the Joined Ontology described in section 7.6.

The *Semantic Recommender System* component forms the system that uses the Joined Ontology and the data stored in *the Domain Triple Stores* to generate recommendations based on Recommendations Equation, illustrated in section 7.7.3. Its sub components are described in section 8.1

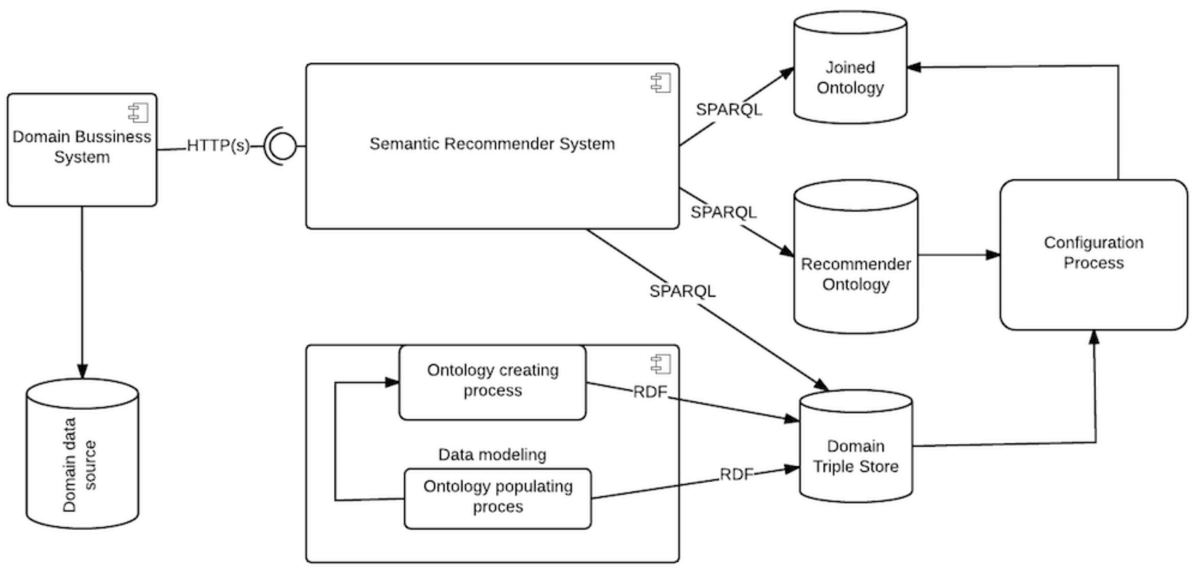


Figure 7-1 Components general overview

1 http://d2rq.org/
2 https://www.w3.org/2005/05/22-SPARQL-MySQL/XTech
3 https://sourceforge.net/projects/rdquery/
4 https://www.w3.org/wiki/DartGrid
5 http://raimond.me.uk/p2r/

7.3 Domain Ontology

It represents a formal definition for the recommendable items, and their characteristics, such as the classes that the items and their features belong to, and the characteristics that define them. It is not necessary required to have all the characteristics of the items, but just those that account for items' similarities, which we call as *Important Predicates* (cf. 7.6.2 for more details) and *Important Classes* (cf. 7.6.3 for more details). In this document, we will use Music Ontology as a domain Ontology.

7.3.1 Domain Ontology Classes

The Music Ontology contains many classes as partially illustrated in Figure 7-2 . It describes the musical pieces (*mo:Item* class) such as operas overtures (*mo:OperaOverture* class), orchestral musical pieces (*mo:OrchestralPieces* class), symphonies (*mo:Symphony* class), sonatas (*mo:Sonata* class) and songs (*mo:Song* class). It also tries to capture more types for the items such as Italian songs (mo:*ItalianSong* class), Verdi's overtures (*mo:VerdiOverture* class), and German's symphonies (*mo:GermanSymphony* class). It captures information about composers (*mo:Composer* class) and artists (*mo:Artist* class). Plus it contains some classifications for the users as German users (*mo:GermanUser* class) and Italian users (*mo:ItalianUser* class).

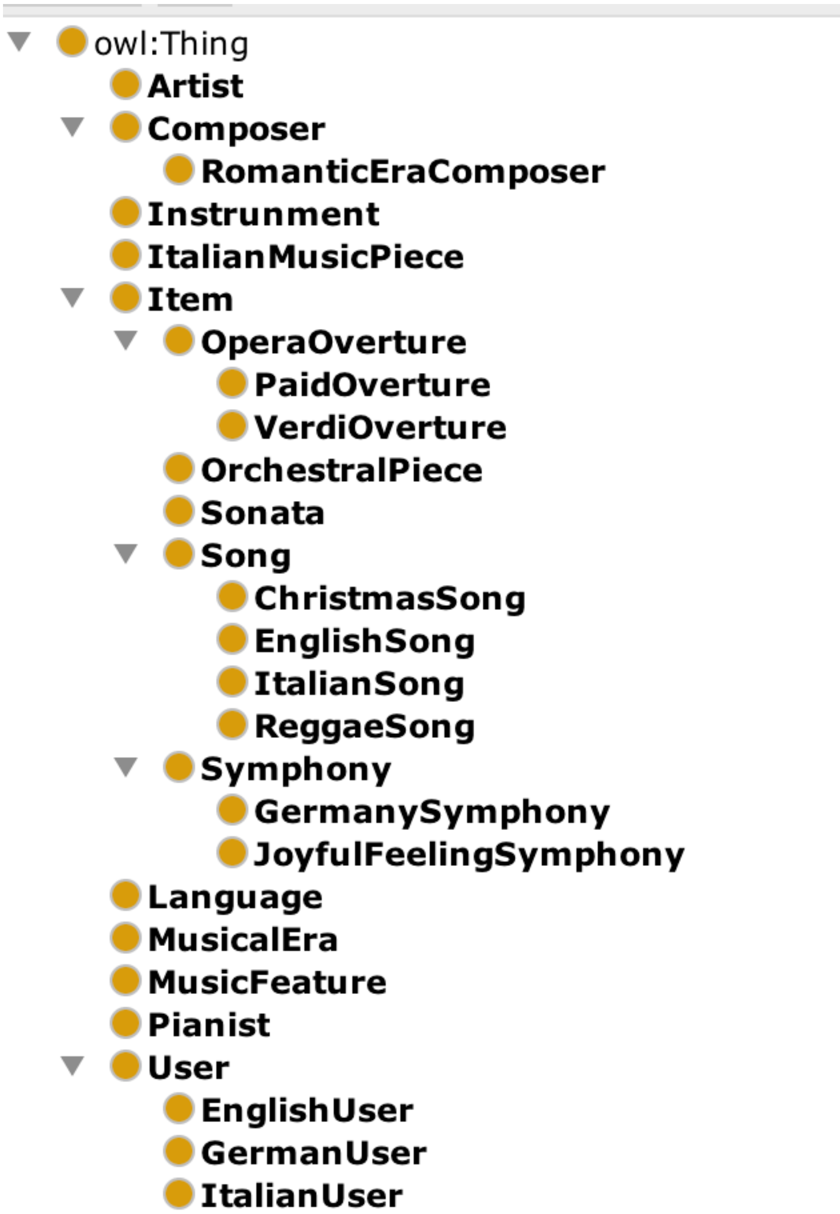


Figure 7-2 Music Ontology Classes

Figure 7-3 contains class diagram for some of the class in the Music Ontology.

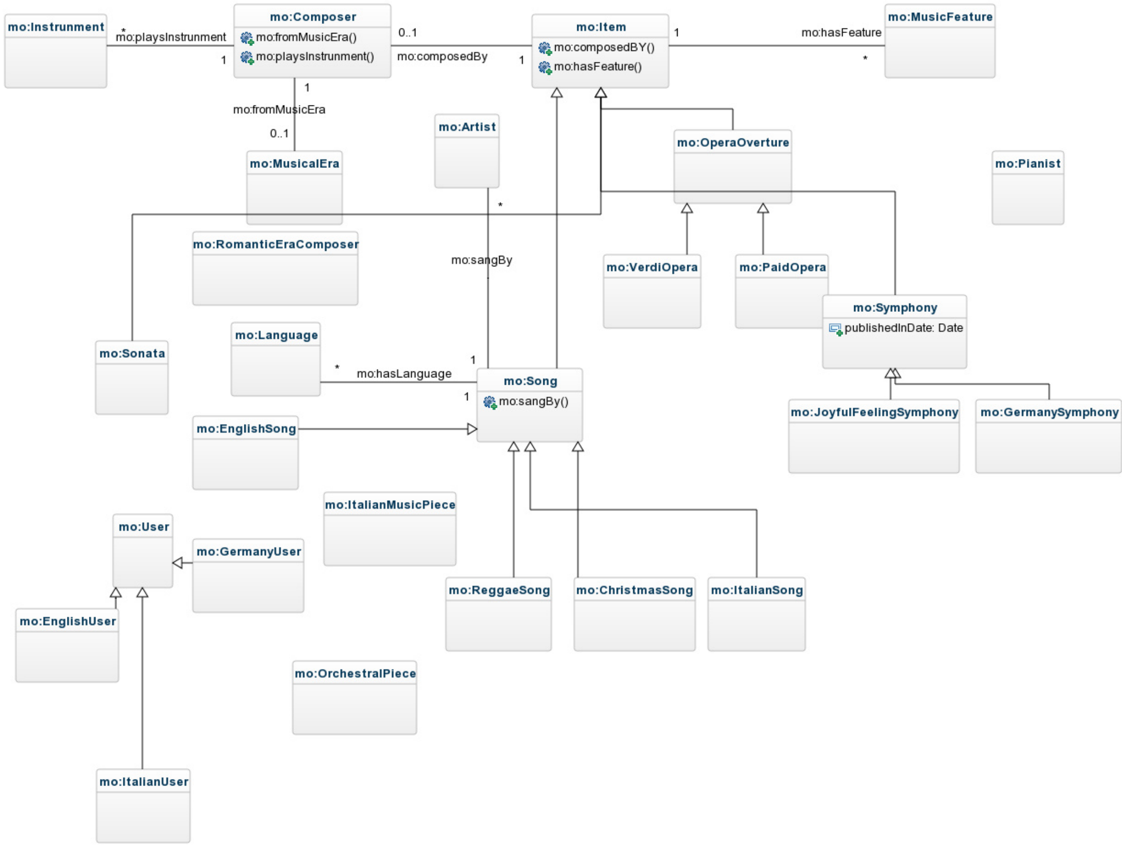


Figure 7-3 Music Ontology class diagram

7.3.2 Domain Ontology Properties

The object properties and data properties in the Music Ontology capture information about the language of the songs (*mo:hasLanguage* predicate), the composers (*mo:composedBy* predicate), their musical era (*mo:fromMusicalEra* predicate), the musical feature of the items (*mo:hasFeature* predicate), and their musical key (*mo:hasKey* predicate).

7.4 Competency Questions

As part of the *Recommender Ontology* development process, the competency questions that underlying the Ontology were identified. The competence questions address both recommendations and business needs. They try to point out the important features of the recommended items and their attributes. They also try to address the fact that some items can be recommended during a specific period, such as Christmas gifts that should be recommended from October until December, while other items, such as event tickets, should never be recommended if the event is over. Plus, they try to capture users' interests and to specify the items that are more important from a business point of view.

1. CQ1

Which are the recommendable items?

2. CQ2

Which are the items' characteristics that are relevant for determining the similarity between two items? And how much each of these characteristics accounts for the similarity?

3. CQ3

Who are the users that the items will be recommended to?

4. CQ4

Which are the items that are more suitable to a specific group of users than other groups? And how does the system alter the similarity in this case?

5. CQ5

Are there items that should never be recommended to specific users?

6. CQ6

Are there items that should be recommended more often to a specific group of users in a specific time period? And how does the system alter the similarity in this case?

7. CQ7

Are there items that should never be recommended to a specific group of users in a specific period?

8. CQ8

What are item characteristics that might reveal users' preferences?

9. CQ9

Whatever the reason is, are there items that should be recommended more often?

10. CQ10

How do users rate the items? And which are the rating values that are considered as liked values?

11. CQ11

Are there features that their importance to items' similarities differs according to the items?

Items features are represented using classes and predicates. However, not all those classes and predicates are important from a similarity point of view. For instance, in the *Music Ontology*, there is a data property *mo:publishedInDate*, but if two items have the same value for that predicate, it does not mean they are similar. While if two items have the same value for *mo:hasFeature* predicate, they should be more similar than two items have different values for the same predicate. Thus, some predicates are important for item similarities while others are not, the same for class, as illustrated in section 7.4.2, and section 7.4.3.

7.4.2 Important Predicates

Not all predicates account for the similarity between items. For example, in the case of *publishedInDate* predicate in the *Music Ontology*, knowing that two symphonies were published in the same date [1] does not give any clue whether or not they are similar. However, the predicate *mo:hasMusicStyle*[2] does account for the similarity since two symphonies having the same musical style should be logically similar.

Moreover, not all Important Predicates account the same for similarities. For example, in the *Music Ontology*, two music pieces share the same value for the Important Predicate *mo:composedBy* predicate are not as similar as two music pieces share the same value for the Important Predicate *mo:hasMusicStyle* predicate. The domain experts are responsible of setting the similarity value for each Important Predicate, and they can do that using the *Recommender Ontology* as will be illustrated later on section 7.5.2.

7.4.3 Important Classes

Not all classes account for the similarity between items. For example, in the *Music Ontology*, knowing that two instances are from class *mo:Symphony* does not give us any clue whether or not they are similar. However, two instances of *mo:JoyfulFeelingSymphony* class should be logically similar.

Moreover, not all Important Classes account the same for similarities. For example, in the *Music Ontology*, two symphonies from the *mo:RomanticEraSymphony* type are not as similar as two symphonies from the *mo:RomanticEraJoyfulFeelingSymphony* type. The

[1] Or in the same year, or the same month …
[2] An example of a music style is Romantic Era Style or Reggae Style.

domain experts are responsible of setting the similarity value for each Important Class, and they can do that using the *Recommender Ontology* as will be illustrated later on section 7.5.3.

7.5 The Recommender Ontology

Figure 7-4, Figure 7-5, and Figure 7-6 show an UML class diagram for the Recommender Ontology.

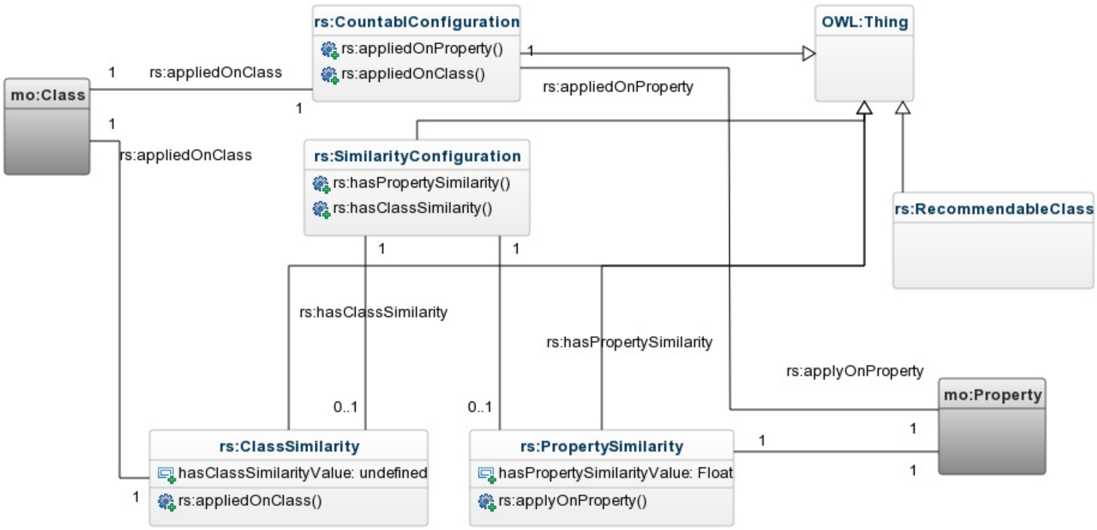


Figure 7-4 Recommender Ontology class diagram (part 1)

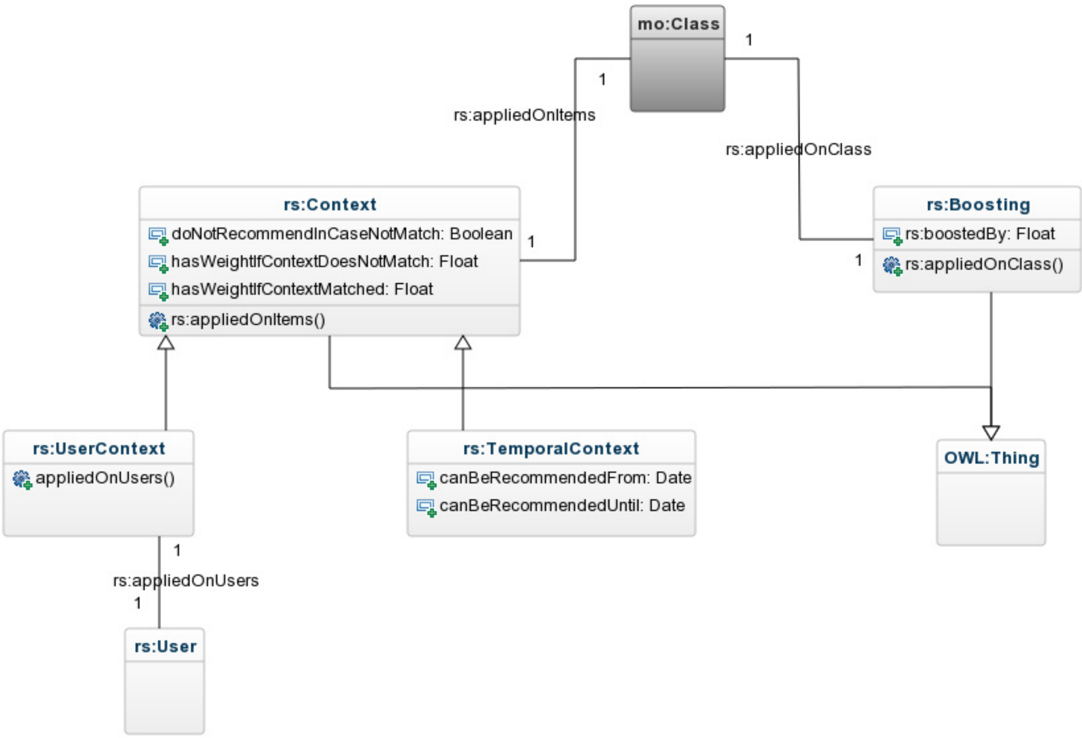


Figure 7-5 Recommender Ontology class diagram (part 2)

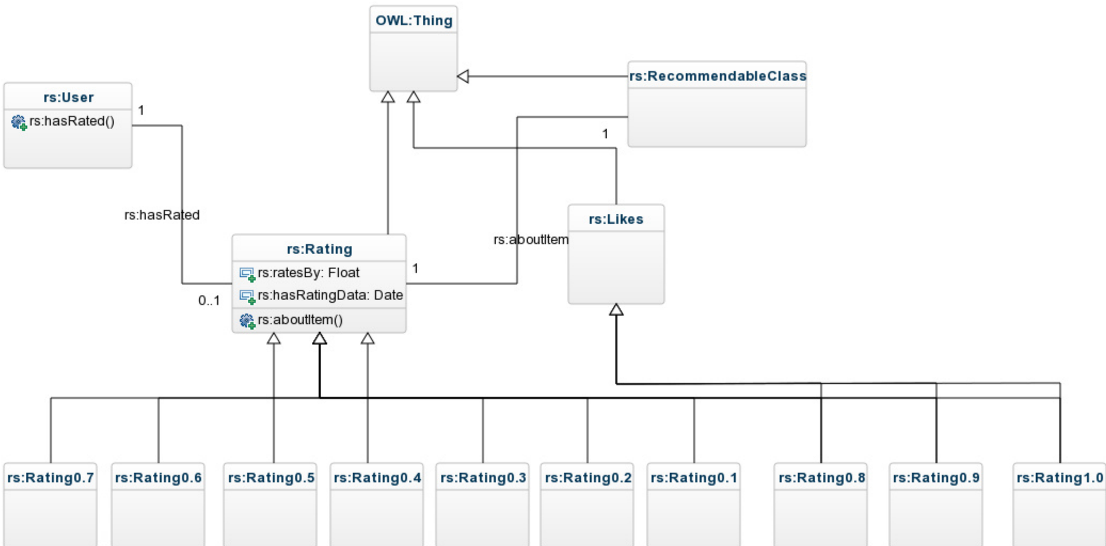


Figure 7-6 Recommender Ontology class diagram (part 3)

7.5.1 Recommendable Class

Competency question number 1 is satisfied by the Recommendable class (*rs:RecommendableClass*) that represents the set of items that can be recommended to the users. Its members (i.e. instances) are defined during the *Configuration* process since such instances are already represented in the domain data triple store. Basically, this consists of annotating one of more classes of the domain Ontology as subclasses of *rs:RecommendableClass* class (cf. 7.6.1 for an example).

7.5.2 PropertySimilarity Class

The Property similarity class (*rs:PropertySimilarity*) is to implement the *Important Predicate* notion (cf. 7.4.2). Each instance of it should have two relationships. The first one is *appliedOnProperty*; its value should be a domain property considered important for similarity purposes. The second one is *hasPropertySimilarityValue*; its value is a float number between 0 and 1 represents how much this relationship accounts for the similarity between the items. The instances of *rs:PropertySimilarity* class and their relationships are defined during the *Configuration* process and they together answer the competency question number 2[1](cf. 7.6.2 for an example).

7.5.3 ClassSimilarity Class

The Class similarity class (*rs:ClassSimilarity*) is to implement the *Important Class* notion, (cf. 7.4.3). Each instance of if should have two relationships. The first one is *rs:appliedOnClass*; its value should be the a domain class considered important for similarity purposes. The second one is rs:*hasClassSimilarityValue*; its value is a float number between 0 and 1 represents how much this class accounts for the similarity between items. The instances of *rs:ClassSimilarity* class and their relationships are defined during the *Configuration* process and they together answer the competency question number 2 (cf. 7.6.3 for an example).

7.5.4 User Class

Competency question number 3 is satisfied by the User class (*rs:User*) that represents the set of users that the items will be recommended to. Its members (i.e. instances) are defined during the *Configuration* process since such users are already represented in the domain data triple store. Basically, this consists of annotating one or more classes of the domain Ontology as subclasses of *rs:User* class(cf. 7.6.4 for an example).

[1] Competency question number 2 is also satisfied by the ClassSimilarity class as illustrated in the next section.

7.5.5 UserContext Class

The User Context class (*rs:UserContext*) is to implement the idea that some items, such as *Italians Songs* in the *Music Domain*, are more suitable for specific users, such as *Italian Users*. Each instance of it should have two relationships; the first one is *rs:appliedOnItems*, which its value is a subclass of the *rs:RecommendableClass* class, and the second one is *rs:appliedOnUsers*, which its value is a subclass of the *rs:User* class. The semantic of this class is: if an item belongs to an Item class, which is specified by the *rs:appliedOnItems predicate*, the system must check first if the active user is from the type specified by the *rs:appliedOnUser* predicate.

Moreover, the instance of the *rs:UserContext* class can have the following predicates:

- *rs:hasWeightIfContextMatched*, which its value is a float number represents an optional weight for the items that match the context in case the active user does as well. For instance, if the value of this predicate is 2 and the item's similarity is 0.75, then the system understands that the users who conform to the context have double weight for this item.
- *rs:hasWeightIfContextDoesNotMatch*, which its value is a float number represents an optional weight for the items that match the context in case the active user does not. For instance, if the value of this predicate is 0.5 and item's similarity is 0.75, the system understands that the users who do not conform to the context have 50% less weight for this item.
- *rs:doNotRecommendInCaseNotMatched*, which its value is a Boolean value, when it is true, the system will not recommend the items to the users who do not conform to the context[1].

The instances of the *rs:UserContext* class and their predicates are defined during the *Configuration* process and they together answer the competency questions numbers 4 and 5 (cf. 7.6.5 for an example.

---

[1] Some many argue that this predicate is not important and a zero value for *hasWeightIfContextDoesNotMatch* predicate could be enough, but in the implementation, an Optional clause was used to check if the item is associated with a UserContext. If not, the clause will not be executed and there will not be any value for the user context filter. Moreover, a Join Graph Patter was used which needs a value for each user context, that means we need to identify the items that do not achieve the Optional condition, and give them a user context value.

7.5.6 TemporalContext Class

Some items can be recommended during a specific period of time, such as the Christmas gifts that can be recommended from October until December. Other items can be recommended until a specific date, such as the tickets for events that can't be recommended if the event is over. Other items can be recommended from a specific date, such as movies that have a published date. Temporal Context class (*rs:TemporalContext*) is to implement this idea. Each instance of it should have a *rs:appliedOnItem* relationship and at least one of *rs:canBeRecommendedFrom* or *rs:canBeRecommendedUntil* relationships. The *rs:appliedOnItem* indicates the items that this context is applied on; its value is a subclass of the *rs:RecommendableClass* class. The *rs:canBeRecommendedFrom* is an OWL data property with *xsd:dateTime* data type range that indicates the starting datetime of the *Recommendation Date Range*[1]. The *rs:canBeRecommendedUntil* is an OWL data property with *xsd:dateTime* data type range that indicates the ending datetime of the Recommendation Date Range.

Moreover, the instance of the *rs:TemporalContext* can have the following predicates:

- *rs:hasWeightIfContextMatched*, which is an OWL data property with *xsd:float* data type range that indicates an optional weight for the items that the context is applied on, if the time in which they are being recommended is inside the Recommendation Date Range. The value should be greater than 1.
- *rs:hasWeightIfContextDoesNotMatch,* which is an OWL data property with *xsd:float* data type range that indicates an optional weight for the items that the context is applied on, if the time in which they are being recommended is not inside the Recommendation Date Range. The value should be less than 1.
- *rs:doNotRecommendInCaseNotMatch,* which is an OWL data property with *xsd:boolean* data type range that if its value is true, the system will not recommend the item in case the recommending time is not inside the Recommendation Date Range.

The instances of the *rs:TemporalContext* class and their relationships are defined during the *Configuration* process and they together answer the competency questions numbers 6 and 7 (cf. 7.6.6 for an example).

[1] The Recommendation Date Range is the date range specified by at least one of *rs:canBeRecommendedFrom* and *rs:canBeRecommendedUntil* predicates.

7.5.7 CountableConfiguration Class

In many domains, users like items that are not similar from content point of view, but they share the same value for a specific attribute. For instance, in a bookstore domain, a user may like the *1984* and the *Animal Farm* novels, which were both written by *George Orwell*. Though the content of these novels are not similar, but we can assume that this user likes *George Orwell* and we could/should recommend more items from *George Orwell*. In this case *George Orwell* is a countable value and *writtenBy* is a countable predicate. Another example would be a singer in a music domain, where users may like different-style songs but performed by the same singer. The Countable Configuration class (*rs:CountableConfiguration*) is to implement this idea. Each instance of it should have two relationships. The first one is *rs:appliedOnProperty* that indicates the property that the system will use to count its value. The second one is rs:*appliedOnClass* that specifies the class of the value that we want to count[1]. The instances of *rs:CountableConfiguration* class and their predicates are defined during the Configuration process and they together answer the competency question number 8 (cf. 7.6.7 for an example).

7.5.8 Boosting Class

In many domains, some items should be recommended more than the others depending on business criteria. For instance, in a content-platform domain, where content providers can publish their content, some providers may have higher priority than the others. This is the case of boosting, where we boost specific items explicitly regarding their content

Boosting Class (*rs:Boosting*) is to implement this idea. Each instance of it should have two predicates. The first one is rs:*appliedOnClass* that indices that class of the items that we want to boost while the second predicate is *rs:boostedBy*, an OWL data property with *xsd:float* range value that specifies the boosting value, which should be greater than 1. The instances of rs:Boosting class with their predicates are defined during the *Configuration* process, and they together answer the competency question number 9. (Cf. 7.6.8 for an example).

[1] We could have not added rs:appliedOnClass predicate, but we decided to add it to handle wrong data. For example, in the Music domain, there is no guarantee that the value of mo:sangBy is an instance of mo:Artist class, we could have a triple like this: mo:stand_by_me_song mo:sangBy mo:americanlFootball . That is a valid RDF triple though we do not want to count mo:americanFootaball because it is obviously a mistake.

7.5.9 Rating Class

The Rating class (*rs:Rating*) is to implement the rating operations. Each instance of it could have three relationships .The first one is *rs:aboutItem* relationship that indicates the item that this rating is about; its object value is an instance of the *rs:RecommendableClass* class. The second relationship is *rs:ratesBy*, which is an OWL data property with *xsd:float* range value that has to be between 0 and 1, and which represents the rating value for the item, specified by *rs:aboutItem* predicates. The third relationship is *rs:hasRatingDate*, which is an OWL data property predicate with *xsd:dateTime* range value specifies the datetime in which the rating has happened.

To specify the user that has done the rating, there is a *rs:hasRated* predicate that its domain is an instance of the *rs:User* class, and its range is an instance of the *rs:Rating* class. The instances of *rs:Rating* class, of its subclasses, which will be illustrated in section 7.5.10, and of the *rs:Likes* class, which will be illustrated in section 7.5.11, with their predicates are defined during the *Configuration* process and they together answer the competency question number 10 (cf. 7.6.9 for an example).

7.5.10 Ratings Subclasses

To handle the 5-stars rating schema, that most of the recommenders use, the *Recommender Ontology* has 10 classes that are subclasses of the rs:*Rating* class, and each one has an OWL value constraints restriction[1] on the rating value, which is the float value of the *rs:ratesBy* predicate. The classes are:

1. *Rating0.1* class represents all the *Rating* instances that have 0.1 as the value their *rs:ratesBy* predicate;
2. *Rating0.2* class represents all the *Rating* instances that have 0.2 as the value their *rs:ratesBy* predicate;
3. *Rating0.3* class represents all the *Rating* instances that have 0.3 as the value their *rs:ratesBy* predicate;
4. *Rating0.4* class represents all the *Rating* instances that have 0.4 as the value their *rs:ratesBy* predicate;
5. *Rating0.5* class represents all the *Rating* instances that have 0.5 as the value their *rs:ratesBy* predicate;

[1] https://www.w3.org/TR/owl-ref/#ValueRestriction

6. *Rating0.6* class represents all the *Rating* instances that have 0.6 as the value their *rs:ratesBy* predicate;
7. *Rating0.7* class represents all the *Rating* instances that have 0.7 as the value their *rs:ratesBy* predicate;
8. *Rating0.8* class represents all the *Rating* instances that have 0.8 as the value their *rs:ratesBy* predicate;
9. *Rating0.9* class represents all the *Rating* instances that have 0.9 as the value their *rs:ratesBy* predicate;
10. *Rating1.0* class represents all the *Rating* instances that have 1.0 as the value their *rs:ratesBy* predicate.

The *Rating* subclasses can be customized according to the business domain. For instance, if the business domain uses a different rating schema, all we need to do is define new *Rating* subclasses, or redefine the already existing ones, with the new business rules. This will not affect the Semantic Recommender because the later uses the *rs:Likes* class, as will be illustrated in section 7.5.11.

7.5.11 Likes class

The Likes class (*rs:Likes*) is to let the system know which are the *Rating* subclasses that the business owners considers as high ratings. For instance in a thumb-up-thumb-down rating schema, the *rs:Likes* class will be the super class for the *thumb-up-Ratings* class, as illustrated in Figure 7-7

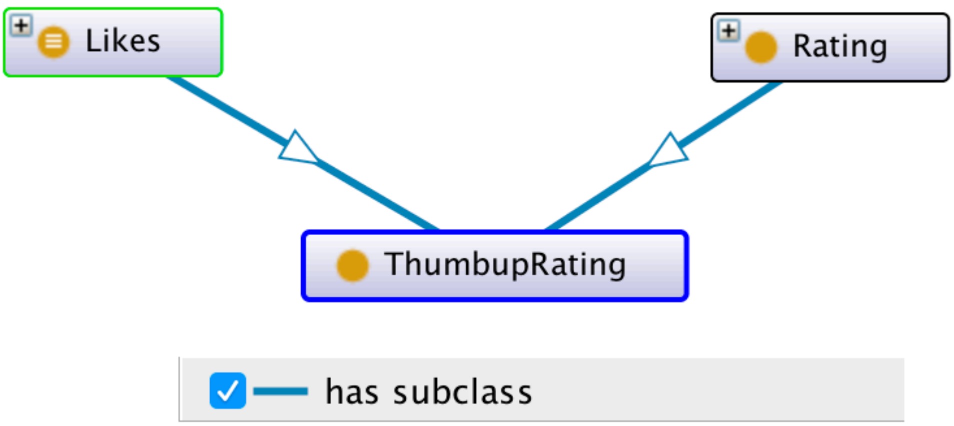


Figure 7-7 Likes example

Another example of using the *rs:Likes* class is to model the purchases. If the business domain offers ratings, we can make the purchases as a subclass or *Rating1.0* class. However, if not, we make the purchases as a subclass of the *Likes* class.

7.5.12 SimilarityConfiguration class

The Similarity Configuration class (*rs:SimilarityConfiguration*) sets all the similarity configurations for a given *rs:RecommendableClass* subclass. In some domains, specific features are more important for some items rather than the others, from similarity point of view. Because the features are captured using predicates and classes, we need to handle the predicates' case and the classes' case differently as illustrated in the next two sub sections.

Setting properties' similarities:

Section 7.5.2 illustrates how to specify formally the Important Properties and their similarity values. However, we still need to state formally which items those properties should be applied to in order to consider them as *Important Properties*. In other words, it is not mandatory the same predicate accounts for the same similarity values always. It could not be important at all for some items, a little bit important for others, and so much important for others. Using *rs:hasPropertySimilarity* predicate can achieve this purpose; where its domain is the *rs:SimilarityConfiguration* class and its range is the *rs:PropertySimilarity* class.

Settings classes' similarities:

Section 7.5.3 illustrates how to specify formally the Important Classes and their similarity values. However, we still need to state formally which items those classes are important to. In other words, we were able to state that if an instance A and an instance B are of an Important Class C, then A and B are similar to some level, but that is not enough, we need to state formally that the class C is just important for specific instances ranther than the others. Using *rs:hasClassSimilarity* predicate can achieve this purpose; where its domain is the *rs:SimilarityConfiguration* class and its range is the *rs:ClassSimilarity* class.

The instances of *rs:SimilarityConfiguration* class with their predicates answers the competency question number 11 and they are defined during the *Configuration* process. (cf. 7.6.10 for an example).

7.6 The Joined Ontology

The *Joined Ontology* is the result of extending and instantiating the *Recommender Ontology* in accordance with a given *Domain Ontology* and a particular business scenario or perspective. Creating it is done manually during the *Configuration* process. The following subsections illustrate the instantiating of the *Recommender Ontology* classes.

7.6.1 Recommendable Items

To select the items that the system should recommend, it is enough to make their class as a subclass of the rs:*RecommendableClass*, which was described in section 7.5.1. For example: *Symphony* class in the *Music Ontology* will be annotated as a subclass or the rs:*RecommendableClass*, as illustrated in the Figure 7-8

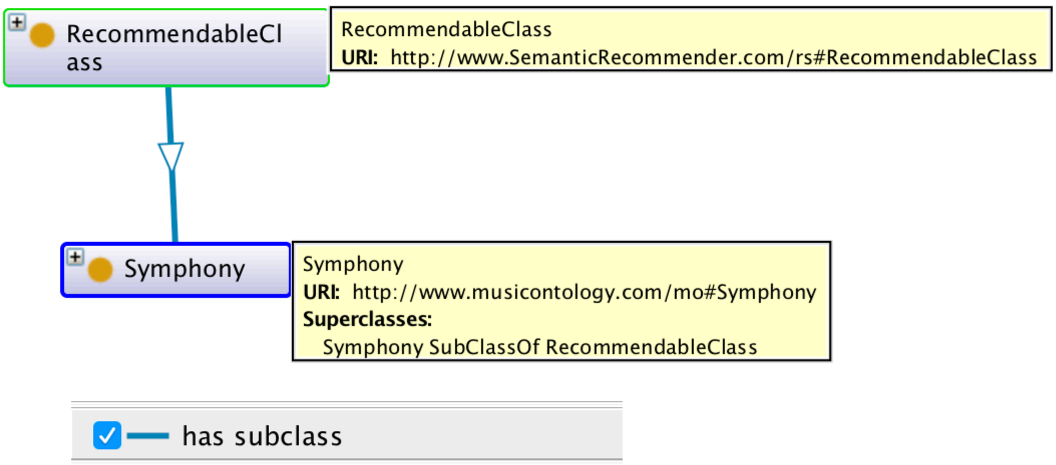


Figure 7-8 RecommendableClass example

7.6.2 Important Predicates

The Joined Ontology has an instance of *rs:PropertySimilarity* Class, which was described in section 7.5.2, for each Important Predicate. For instance, we assume that in the *Music Ontology* the experts decide that if two symphonies share the same feature, these two symphonies are 50% similar. Thus, in the *Joined Ontology*, there will be an instance of the rs:*PropertySimilarity* class, *hasFeaturePropertySimilarity*, that has *hasFeature* property as the object of the *appliedOnProperty* predicate, and has 0.5 as the object of *hasPropertySimilarityValue* predicate, as illustrated in Figure 7-9

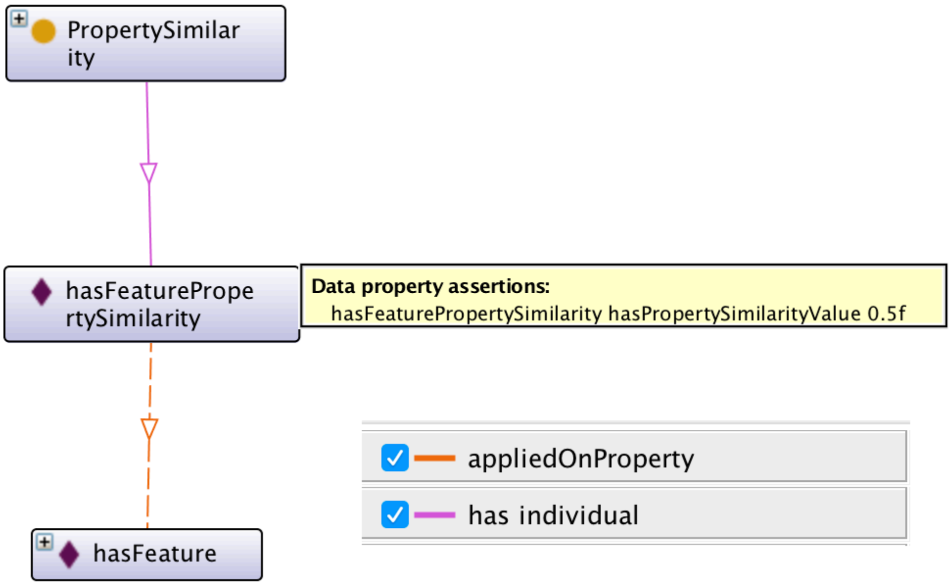


Figure 7-9 PropertySimilarity example

The same example can be written in Turtle format [1] as:

http://www.musicsemanticontology.com/mso#hasFeaturePropertySimilarity

```
:hasFeaturePropertySimilarity rdf:type owl:NamedIndividual , rs:PropertySimilarity ;
 rs:appliedOnProperty  mo:hasFeature ;
 rs:hasPropertySimilarityValue "0.5"^^xsd:float .
```

7.6.3 Important Classes

The *Joined Ontology* has an instance of *rs:ClassSimilarity* class, which was described in section 7.5.3, for each Important Class. For example, we assume that in the *Music Ontology* the experts decide that if two symphonies are from type *JoyfulFeelingSymphony*, they are 75% similar. Thus, in the *Joined Ontology*, there will be an instance of the *rs:ClassSimilarity* class, *joyfulFeelingSymphonyClassSimilarity*, that has *JoyfulFeelingSymphony* class as the object of the predicate *rs:appliedOnClass*, and has 0.75 as the object of the *rs:hasClassSimilarityValue* predicate, as illustrated in Figure 7-10

[1] We will use a screenshot for the triples rather than a Turtle format RDF triples because we believe it is easier to understand

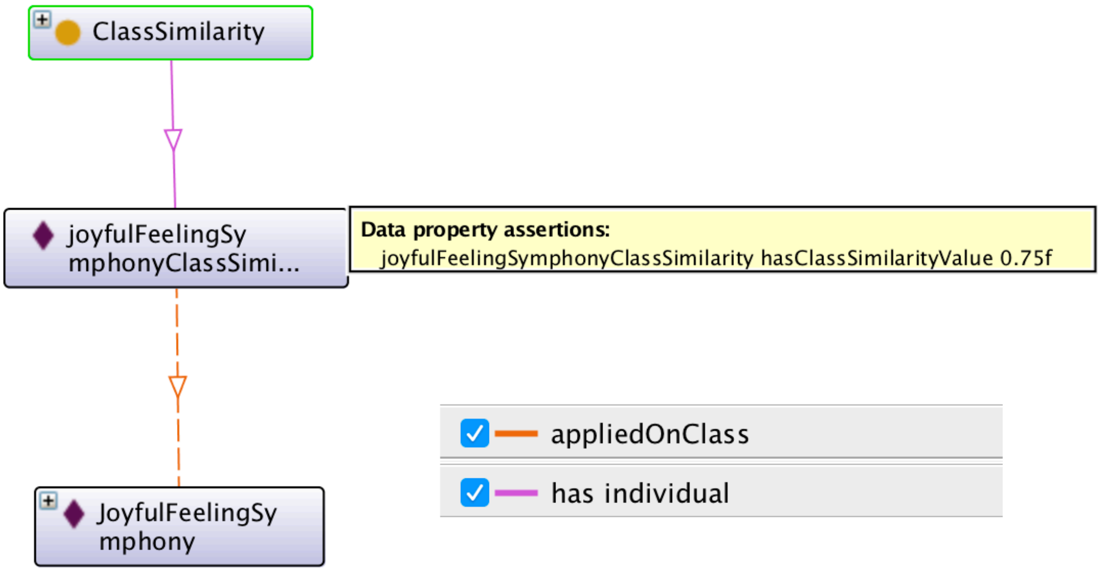


Figure 7-10 ClassSimilarity example

7.6.4 Users

To specify the users that the system will recommend items to, it is enough to make the *User* class from the *Domain Ontology* as a subclass of the *rs:User* class, which was described in section 7.5.4. For instance, Figure 7-11 shows how the *User* class from the *Music Ontology* is a subclass of the *rs:User* class of the *Recommender Ontology*.

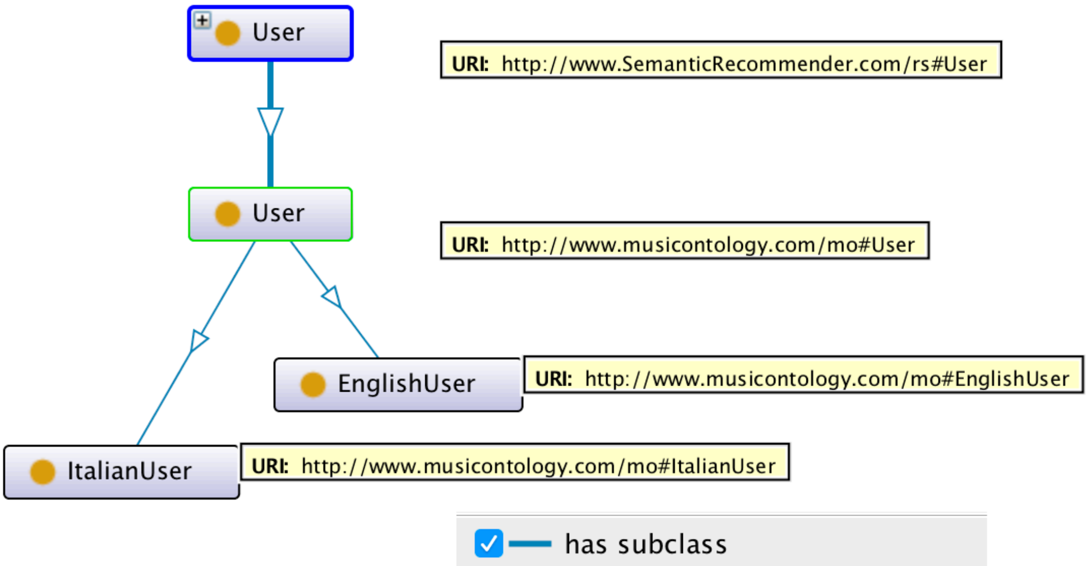


Figure 7-11 User class example

7.6.5 User Context

The *Joined Ontology* has an instance of *rs:UserContext* class, which was described in section 7.5.5, for each User Context. For instance, in the *Music* domain, the *Italian Music Pieces* are two times important for the people who speak Italian. This can be done by specifying a *User Context* instance, *italianMusicPieceItalianUserContext*, that has *ItalianMusicPiece* class as the object value for rs:*appliedOnItem* predicate, and has *ItalianUser* class as the object value for rs:*appliedOnUser* predicate, as illustrated in Figure 7-12

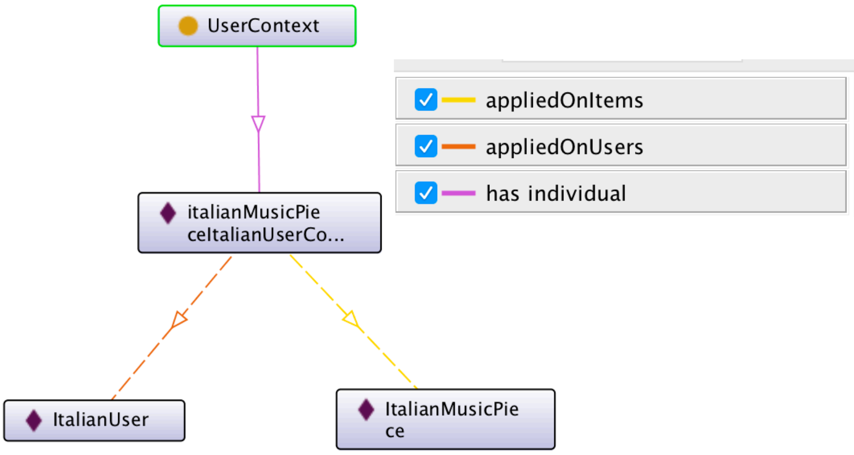


Figure 7-12 UserContext example

7.6.6 Temporal Context

The *Joined Ontology* has an instance of *rs:TemporalContext* class, which was described in section 7.5.6, for each Temporal Context. For example, in the Music domain, some German symphonies should have higher weight during the Annual German Symphony Festival. This can be done by specifying a *rs:TemporalContext* instance, *symphonyFestival2016*, that has *GermanSymphony* class as the object of the *rs:appliedOnItem* predicate, and has June, 1st, 2016 as the value of *rs:canBeRecommendedFrom* predicate, and June, 30th, 2016 as the value of the *rs:canBeRecommendedUntil* predicate, as illustrated in Figure 7-13

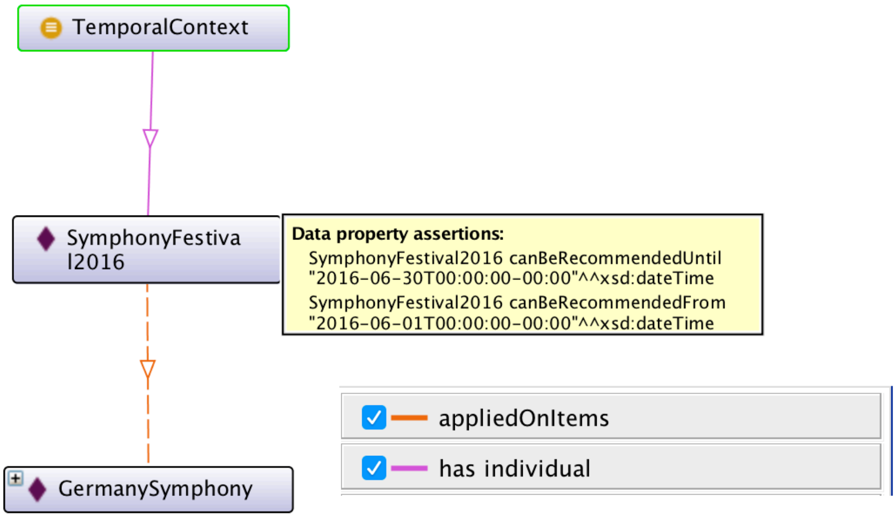


Figure 7-13 TemporalContext example

An example about not recommending an item after a specific date would be the case in which the system is recommending tickets for an event; if the event is over, the system must not recommend the tickets any more. This can be done by creating an instance of the *TemporalContext*, and set true to its *doNotRecommendInCaseNotMatch* predicate, as illustrated in Figure 7-14

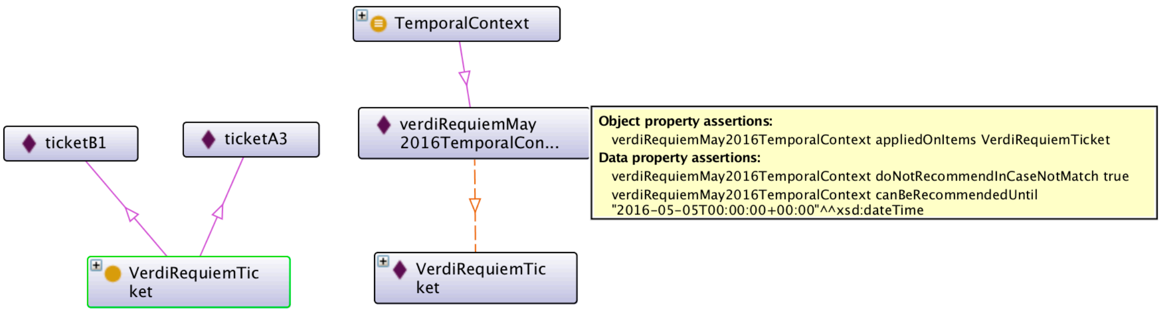


Figure 7-14 Verdi Requiem Temporal Context

7.6.7 Countable Class

The *Joined Ontology* has an instance of *rs:CountableConfiguration* class, which was described in section 7.5.7, for each value that the domain business owner wants to count on the liked items. For instance, in case we wanted to apply the *Countable* principle on the composers that the active user has liked symphonies composed by, we create an instance of the *rs:CountableConfiguration* class, *countableComposer*, that has *rs:composedBy* predicate as the object value of the *rs:appliedOnProperty* predicate and has *Composer* class as the object value of the *rs:appliedOnClass* predicate, as illustrated in Figure 7-15

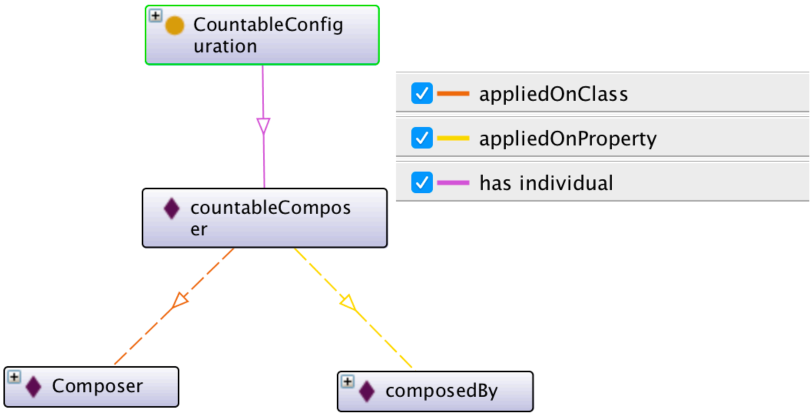


Figure 7-15 CountableConfiguration example

7.6.8 Boosting Class

The *Joined Ontology* has a Boosting instance, which was described in section 7.5.8, for each group of items that the business owner want to boost. For example, in the *Music* domain, to boost the opera overtures, we can create an instance of the *rs:Boosting* class, *operaOvertureBoosting*, with *OperaOverture* class as the object value of its *rs:appliedOnClass* predicate and 2 as the value of its *rs:boostedBy* predicate, as illustrated in Figure 7-16

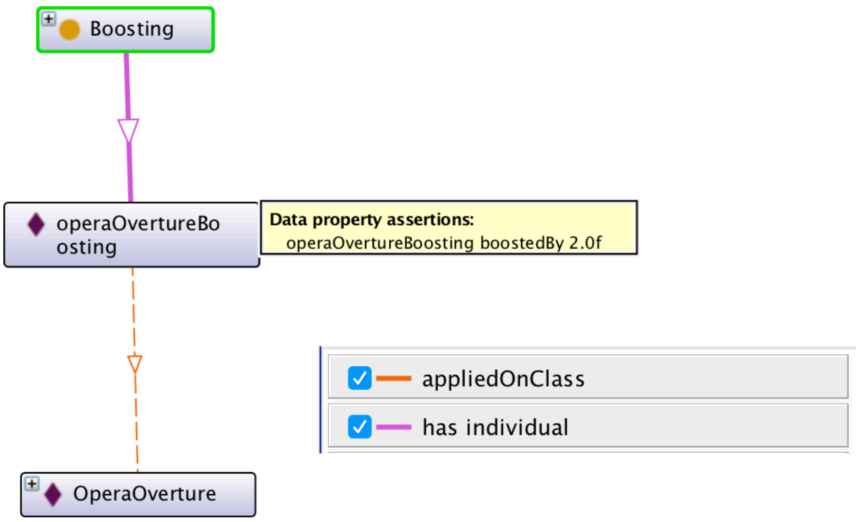


Figure 7-16 Boosting example

7.6.9 Rating Class

The *Joined Ontology* contains an instance of the *rs:Rating* class, which was described in section 7.5.9, for each rating the users do. For instance, if the user *Galileo Galilei* rates *9th_symphony_for_beethoven* by 0.9, the *Joined Ontology* will have an instance of the *Rating* class, *galileoGalilei_RatingFor9thSymphonForBeethoven*, that has *9th_symphony_for_beethoven* as the object of the *rs:aboutItem* predicate, and has 0.9 as the value of the data property *rs:ratesBy*, as illustrated in Figure 7-17. To state that the mentioned Rating's instance belongs to the user *Galileo Galilei*, the user's instance *Galileo Galilei* in the *Joined Ontology* will have *galileoGalilei_RatingFor9thSymphonForBeethoven* as the value of the predicate *rs:hasRated*, as illustrated in Figure 7-18

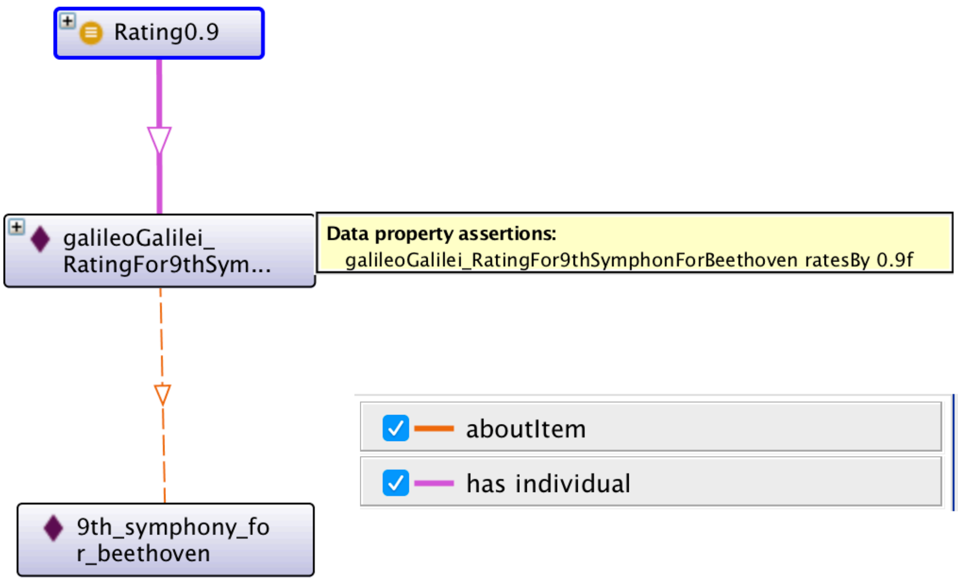


Figure 7-17 Galileo Galilei rating for 9th symphony for Beethoven

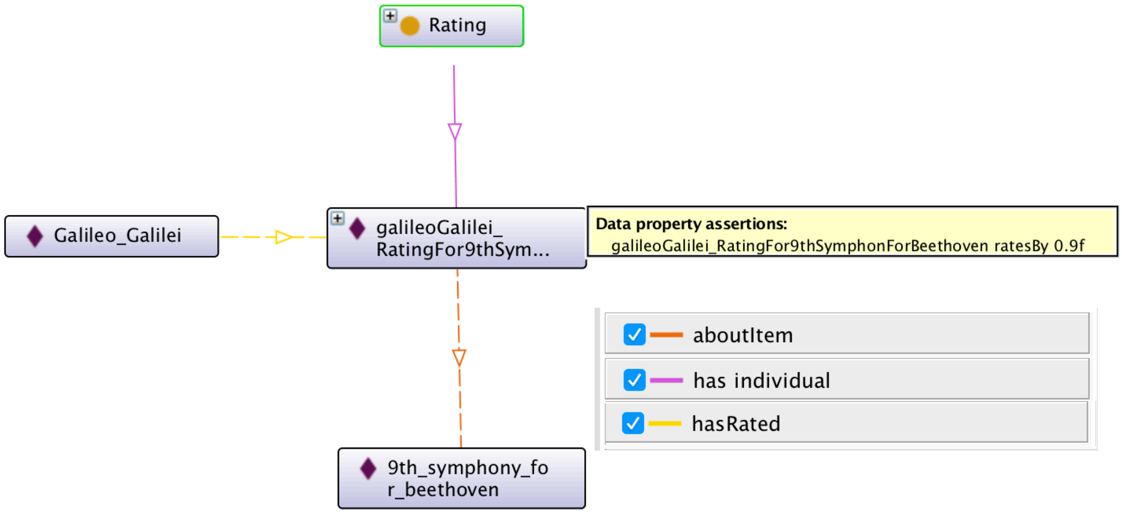


Figure 7-18 Galileo Galilei rating for 9th symphony for Beethoven complete

7.6.10 SimilarityConfiguration class

The *Joined Ontology* contains at least one instance of the rs:*SimilarityConfiguration* class, which was described in section 7.5.12. If the domain experts decide that some features are more important for some items rather than the others, this can by done be defining two instances of the *rs:SimilarityConfiguration* class, each one captures the degree of important for those features.

Setting properties' similarities:

Section 7.6.2 states formally that the similarity of *hasFeature* predicate is 50 percent. However, we still need to state formally that the *hasFeature* predicate is applied on the *Symphony* class and not on any other class. This can be done using the *rs:SimilarityConfiguration* class, *rs:hasSimilarityConfiguration* predicate, and the *rs:hasPropertySimilarity* predicate, as illustrated in Figure 7-19. The *musicSimilarityConfiguration* instance is from type *rs:SimilarityConfiguration* and has the previously created *hasFeaturePropertySimilarity* instance as the object value of the *rs:hasPropertySimilarity* predicate. Then the *hasSimilarityConfiguration* predicate is used to link the *Symphony* class with the *musicSimilarityConfiguration* instance.

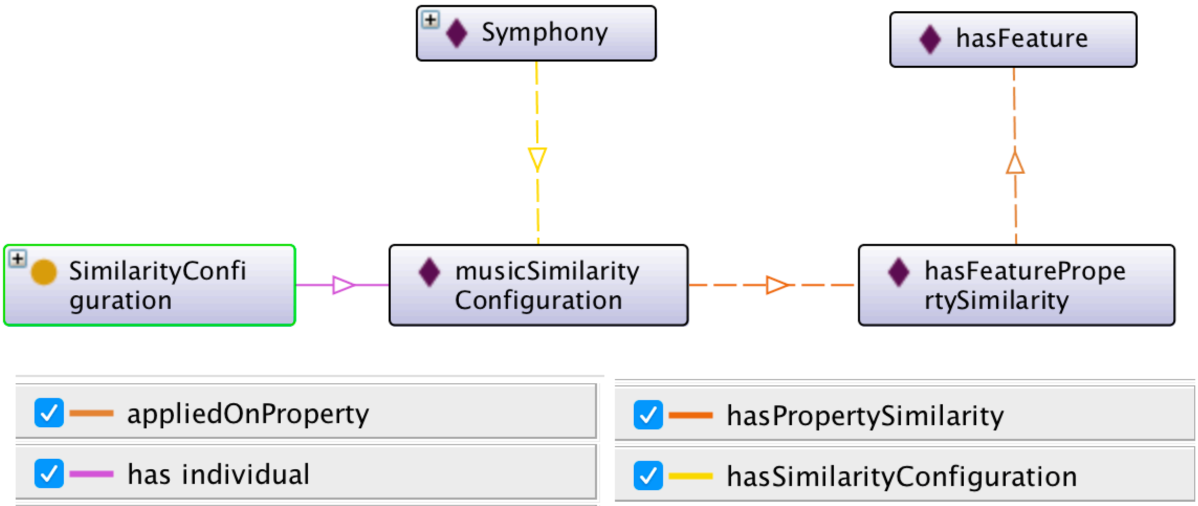


Figure 7-19 SimilarityConfiguration on PropertySimilarity

Using that approach allows us to set different similarity values for the same predicate depending on the object that predicate is applied on. For instance, in an online shop store, we may need to say that if two books have the same genre, they are 20% similar, while if two shirts have the same genre, they are 50% similar. Thus, the *hasGenre* predicate has 0.2 similarity value if it is applied on *Book*, and 50% similarity if it is applied on *Shirts*. To state that formally, we create two instances of the *rs:SimilarityConfiguration* class, which are *bookSimilarityConfiguration* and *shirtSimilarityConfiguration*, and link each one of them, using *hasPropertySimilarity* predicate, to an instance of the *rs:PropertySimilarity* class and which has the needed similarity value. Then we link the *Book* class to the *bookSimilarityConfiguration* and the *Shirts* class to the *shirtSimilarityConfiguration* using *hasSimilarityConfiguration* predicate, as illustrated in Figure 7-20

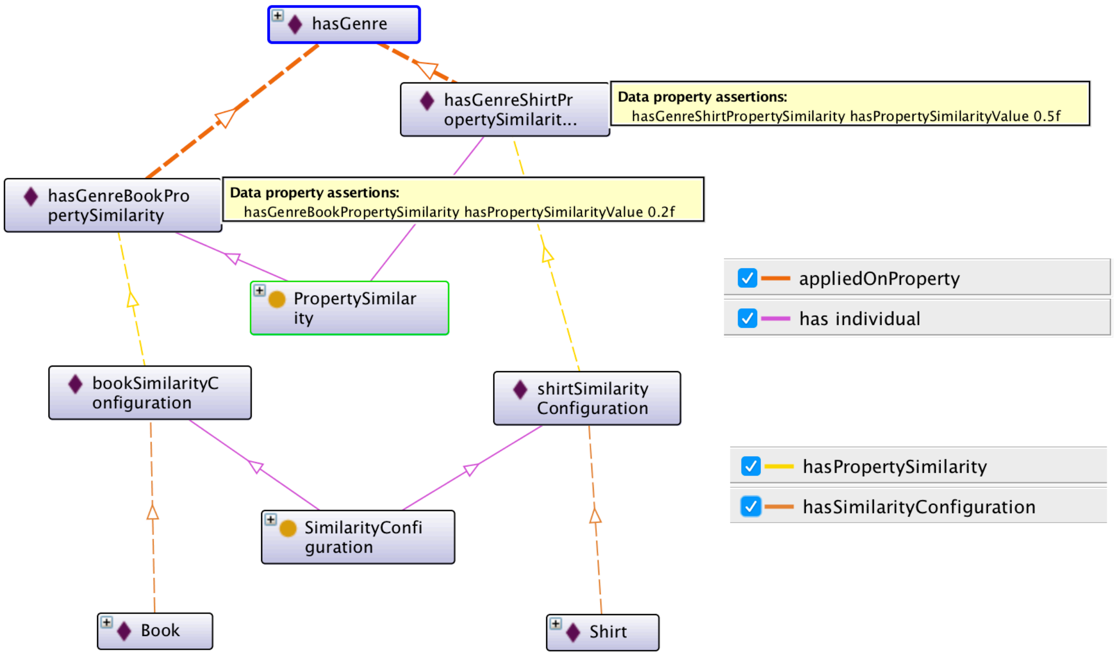


Figure 7-20 Same property different similarity values

Settings classes' similarities:

Section 7.6.3 states formally that the similarity between two instances of the *JoyfulSymphony* class is 75%. However, we still need to state formally that the two instances must be two <u>symphonies</u>. In other words, this *ClassSimilarity* is applied on the *Symphony* class. This can be done using the *rs:SimilarityConfiguration* class, the *hasSimilarityConfiguration* predicate, and the *rs:hasClassSimilarity* predicate, as illustrated in Figure 7-21. The *musicSimilarityConfiguration*, created in the previous paragraph, will have the *joyfulFeelingClassSimilarity* as the object value of the *rs:hasClassSimilarity* predicate [1].

[1] The Symphony class is already linked to the musicSimilarityConfiguration instance from the previous paragraph example.

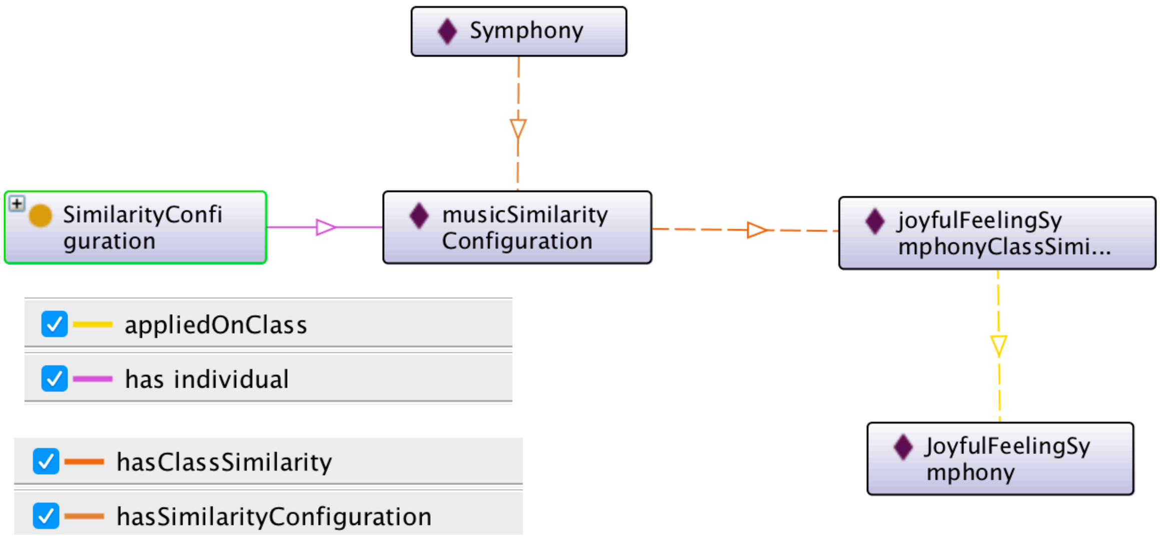


Figure 7-21 SimilarityConfiguration on ClassSimilarity

Using that approach allows us to set different class similarities for the same class depending on the class of the item. For instance, in an online shop store, we may need to say that if two books are from *LoveMaterial* class, they are 30% similar, but if two movies are from *LoveMaterial* class, they are 60% similar. Thus, the class similarity value for the *LoveMaterial* is 0.3 in the case of books and 0.6 in the case of movies. To state that formally, we create two instances of the *rs:SimilarityConfiguration* class, which are *bookSimilarityConfiguration* and *movieSimilarityConfiguration*, and link each one of them, using the *hasClassSimilarity* predicate, to an instance of the *ClassSimilarity* class, and which has the needed value for the similarity. Then we link the *Book* class to the *bookSimilarityConfiguration* and the *Movie* class to the *movieSimilarityConfiguration* using the *hasSimilarityConfiguration*, as illustrated in Figure 7-22

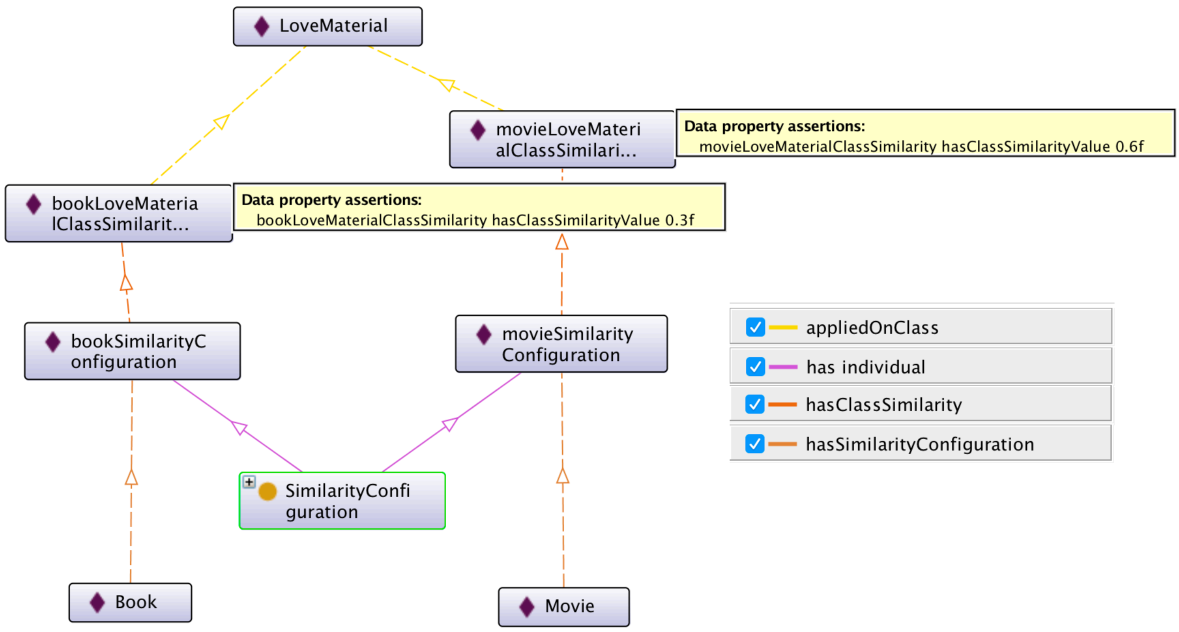


Figure 7-22 Same class different similarity values

7.7 Calculating the similarity

The data that is modeled using RDF forms a graph. Basically we are exploring that graph to find recommendations. The similarity evaluation introduces the notion of levels; we will use two levels, Level 0, illustrated in section 7.7.1, and Level 1 illustrated in section 7.7.2.

The Semantic Recommender generates recommendations following the three steps below:

1. Suggest items similar to that item depending on *Level 0* and *Level 1* as described in sections 7.7.7 and 7.7.10
2. Filter these items according to the *User Context* and *Temporal Context* as described in sections 7.7.14 and 7.7.16
3. Boost these items according to the *Boosting* as described in section 7.7.18

7.7.1 Level 0

Instance Case:

Two items are from Level 0 if they share the same value for the same *Important Predicate*. For instance, in the *Music Ontology*, the *9th_symphony_for_beethoven* and the *new_world_symphony* instances are from Level 0 because they share the same object, which is *compelling_intensity*, for the *Important Predicate hasFeature*, as illustrated in Figure 7-23

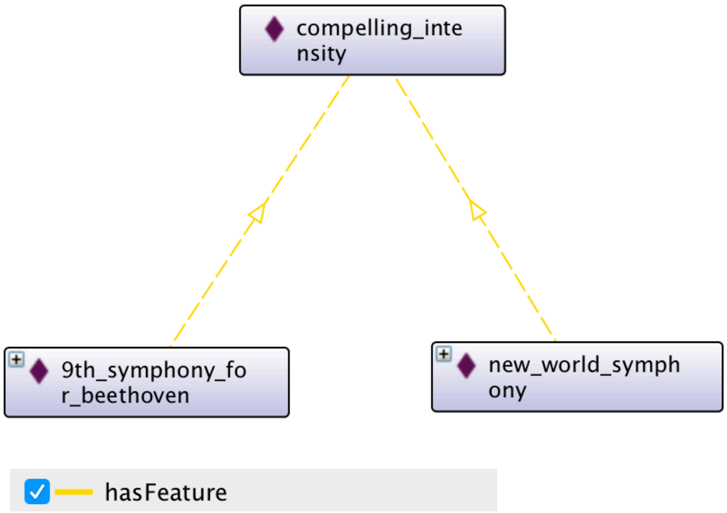


Figure 7-23 Level0 instance case

Class Case:

Two items are from Level 0 if they are instances from the same *Important Class*. For instance, in the *Music Ontology*, the *9th_symphony_for_beethoven* and the *new_world_symphony* instances are from Level 0 because they are both an instance from *JoyfulFeelingSymphony* class, as illustrated in Figure 7-24

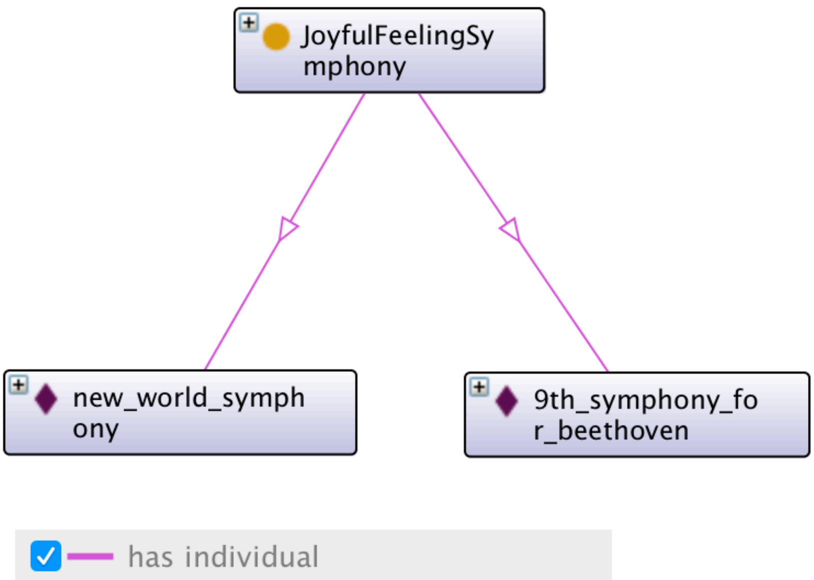


Figure 7-24 Level0 class case

7.7.2 Level 1

Instance Case:

Two items are from Level 1 if they have different values for the same *Important Predicate*, but these two values share the same value for an *Important Predicate*. For instance, the *9th_symphony_for_beethoven* and the *new_world_symphony* instances are from Level 1 because they have different values for the *Important Predicate composedBy*, which are *Beethoven* and *Dvorak* respectively, but these values share the same instance, which is *romantic_era* for the *Important Predicate fromMusicalEra*, as illustrated in Figure 7-25

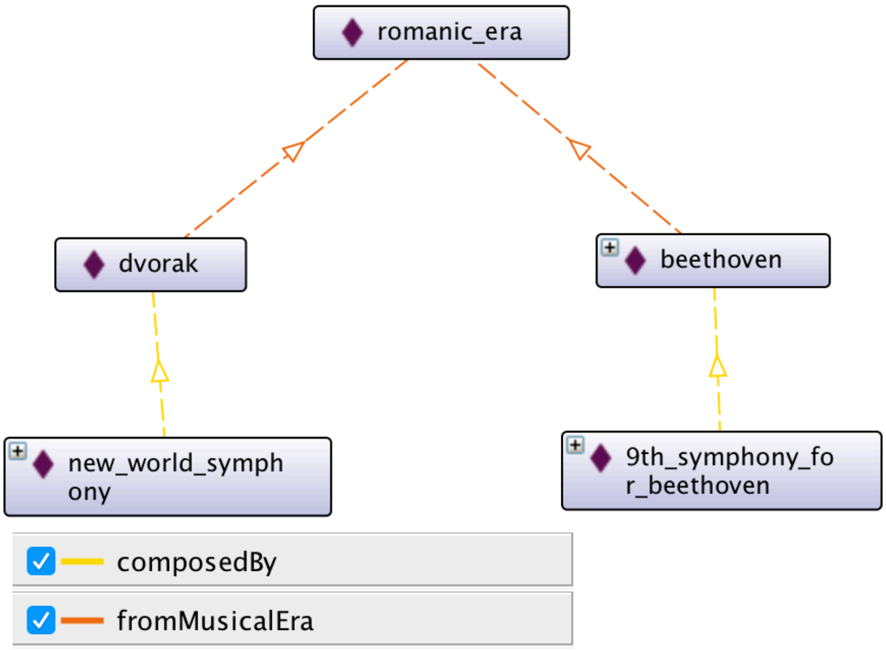


Figure 7-25 Level1 instance case

Class Case:

Two items are from Level 1 if they share different values for the same *Important Predicate*, but these two values are instances from an *Important Class*. For example, the *9th_Symphony_for_Beethove* and the *new_world_symphony* instances are from Level 1 because they have different values for the *Important Predicate composedBy*, which are *Beethoven* and *Dvorak* respectively, but these values are instances from the *Important Class Pianist*, as illustrated in Figure 7-26

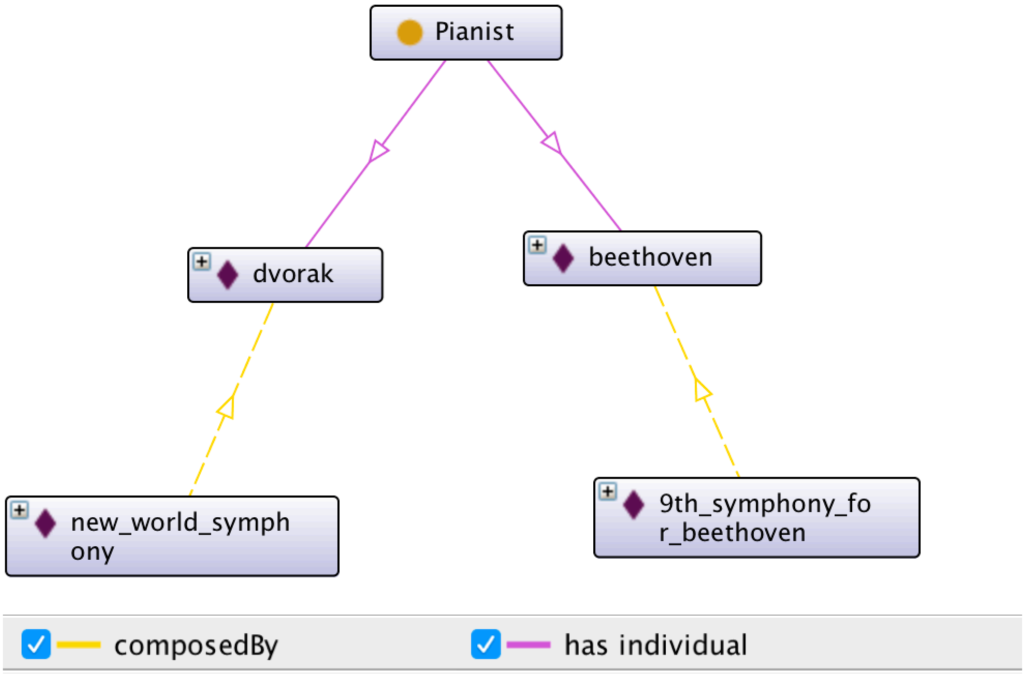


Figure 7-26 Level1 class case

7.7.3 Recommendation Equation

The recommendations for an active user (u) is generated according to the Recommendation Equation 21

Equation 21 Recommendations Equation

$$Recommendation(u) = \sum_{i}^{All\ items\ the\ active\ user\ has\ liked} \sum_{j}^{All\ items\ in\ level\ 0\ and\ level\ 1\ for\ the\ liked\ item\ (i)} Sim(i,j,u)$$

$$u \in U : U\ is\ the\ set\ of\ all\ users$$

$$i,j \in I : I\ is\ the\ set\ of\ all\ items$$

$Sim(i,j,u)$ is the similarity equation illustrated in section 7.7.4.

7.7.4 Similarity Equation

The similarity between an item A and another item B for user u is calculated using this equation:

Equation 22 Similarity Equation

$$sim(i,j,u) = LevelSimilarities(i,j,u) \times UserContextWeight(j,u) \times TemporalContextWeight(j,t) \times BoostingWeight(j)$$

Where:

- $LevelSimilarities(i,j,u)$ Function is illustrated in Equation 23,

- $UserContextWeight(j,u)$ Function is illustrated in Equation 25,
- $TemporalContextWeight(j,t)$ Function is illustrated in Equation 26,
- $BoostingWeight(j)$ Function is illustrated in Equation 27.

Equation 23 Level Similarities Equation

$$LevelSimilarities(i,j,u) = PureSimilarities(i,j) \times Rating(i,u)$$

Where $Rating(i,u)$ function is the rating of the user (u) to the item (i), and $PureSimilarities(i,j)$ function is illustrated in Equation 24.

Equation 24 Pure Similarity Equation

$$\begin{aligned} PureSimilarities(i,j) &= \sum_{p}^{All\ Important\ Predicates} PropertySimilarity(p) \times LevelImportant(l) \\ &\times InstanceImportance() \\ &+ \sum_{c}^{All\ Important\ Classes} ClassSimilarit(c) \times LevelImportance\ (l) \\ &\times ClassImporance() \end{aligned}$$

Where:

- $PropertySimilarity(p)$ Function is the similarity value for the Important Property (p)
- $LevelImportanc(l)$ Function is *Level Importance* value for the level l, which is a static value representing the importance of that level. We suggest that if two items are from *Level 0*, they are 2 times more similar than two items from *Level 1*. So the *Level Importance* value for Level 0 is 2/3, while the *Level Importance* value for Level 1 is 1/3.
- $InstanceImportance()$ Function is the *Instance Importance* value and $ClassImportance()$ Function is the *Class Importance* value. For each level, the system expects a value for *Class Importance* and *Instance Importance*, which are static values representing the importance of class level case and instance level case, that are described in 7.7.1 and 7.7.2, because in some domains, the instance level is more important than the class level, and vice versa in other domains. In other words, in some domain, have two instances that share the same value for the same predicate are considered more similar than two instances from the same type.
- $ClassSimilarity(c)$ Function is the similarity value for the Importance Class (c)

Examples about LevelSimilarities including Level 0 instance case is illustrated in section 7.7.5, Level 0 class case is illustrated in section 7.7.6, Level 0 both instance and class cases is illustrated in section 7.7.7, Level 1 instance case is illustrated in section 7.7.8, Level 1 class case is illustrated in section 7.7.9, Level 1 both instance and class cases is illustrated in section 7.7.10, and both level 0 and Level 1 is illustrated in section 7.7.11.

Equation 25 User Context weight equation

$$UserContextWeight(j,u) = \sum_{ux}^{All\ User\ Contexts} SingleUserContextWeight(ux,j,u)$$

Where $SingleUserContextWeight(ux, j, u)$ function is the weight of the User Context (ux) against both the item (j) and the active user (u), as illustrated in section 7.6.5.

Examples of User Contexts similarities are illustrated in sections 7.7.13 and 7.7.14.

Equation 26 Temporal Context weight equation

$$TemporalContextWeight(j, t) = \sum_{tx}^{All\ Temporal\ Contexts} SingleTemporalContextWeight(tx, j, t)$$

Where $SingleTemporalContextWeight(tx, j, t)$ function is the weight of the Temporal Context (tx) against the item (j) and the current date time (t), as illustrated in section 7.6.6.

Examples of Temporal Contexts similarities are illustrated in sections 7.7.15 and 7.7.16.

Equation 27 Boosting weight equation

$$BoostingWeight(j) = \sum_{b}^{All\ Boostings\ that\ applied\ on\ j} SingleBoostingWeight(b, j)$$

Where $SingleBoostingWeight(b, j)$ function is the weight of the boosting instance (b) against the item (j), as illustrated in section 7.6.8.

An example of Boosting similarities is illustrated in sections 7.7.18.

7.7.5 Level 0 Instance Similarity

Section 7.7.1 shows an example of two instances, *9th_symphony_for_beethoven* and *new_world_symphony* from Level 0 sharing the same value for the Important Predicate *hasFeature*, and section 7.6.2 states that *hasFeature* predicate has a 50% similarity value. Applying the *Similarity Equation*, described in section 7.7.4, then the similarity between them is calculated as below (we don't have any user contexts nor temporal context, so basically it is *LevelSimilarities*):

$$PropertySimilarityLevel0 \times Level0Importance \times InstanceImportance$$

$$Sim = \frac{50}{100} \times \frac{2}{3} \times 1 \approx 0.33$$

However, that is just the pure similarity between the two items. We still need to customize it according to the user's preference. As a result, if the user *Galileo Galilei* rates *9th_symphony_for_beethoven* by 0.9, then the final similarity is:

$$PropertySimilarityLevel0 \times Level0Importance \times InstanceImportance * rating$$

$$Sim = 0.33 \times 0.9 = 0.3$$

7.7.6 Level 0 Class Similarity

Section 7.7.1 shows an example of two instances, *9th_symphony_for_beethoven* and *new_world_symphony*, from the same type, *JoyfulFeelingSymphony* and section 7.6.3 states that *JoyfulFeelingSymphony* has 75% similarity value. Applying the Similarity Equation, described in section 7.7.4, then the similarity between them is calculated as:

$$ClassSimilarityLevel0 \times Level0Importance \times ClassImportance$$

$$Sim = \frac{75}{100} \times \frac{2}{3} \times \frac{5}{10} = 0.25$$

However, that is just the pure similarity between the items. We still need to customize it according to the user's preference. As a result, if the user *Galileo Galilei* rates *9th_symphony_for_beethoven by* 0.9[1], then the final similarity is

$$ClassSimilarityLevel0 \times Level0Importance \times ClassImportance \times rating$$

$$Sim = 0.25 \times 0.9 = 0.225$$

7.7.7 Level 0 both instance and class similarity

Figure 7-27 is the result of combing the both examples provided in section 7.7.1, applying the Similarity Equation, described in 7.7.4, the similarity between *9th_symphony_for_beethoven* and *new_world_symphony* is calculated as:

$$PropertySimilarityLevel0 \times Level0Importance \times InstanceImportance + ClassSimilarityLevel0 \times Level0Importance \times ClassImportance$$

$$Sim = \frac{50}{100} \times \frac{2}{3} \times 1 + \frac{75}{100} \times \frac{2}{3} \times \frac{5}{10} \approx 0.58$$

[1] We will assume, in all the next examples, that Galileo Galilei's rating for the 9th_symphony_for_beethoven is 0.9

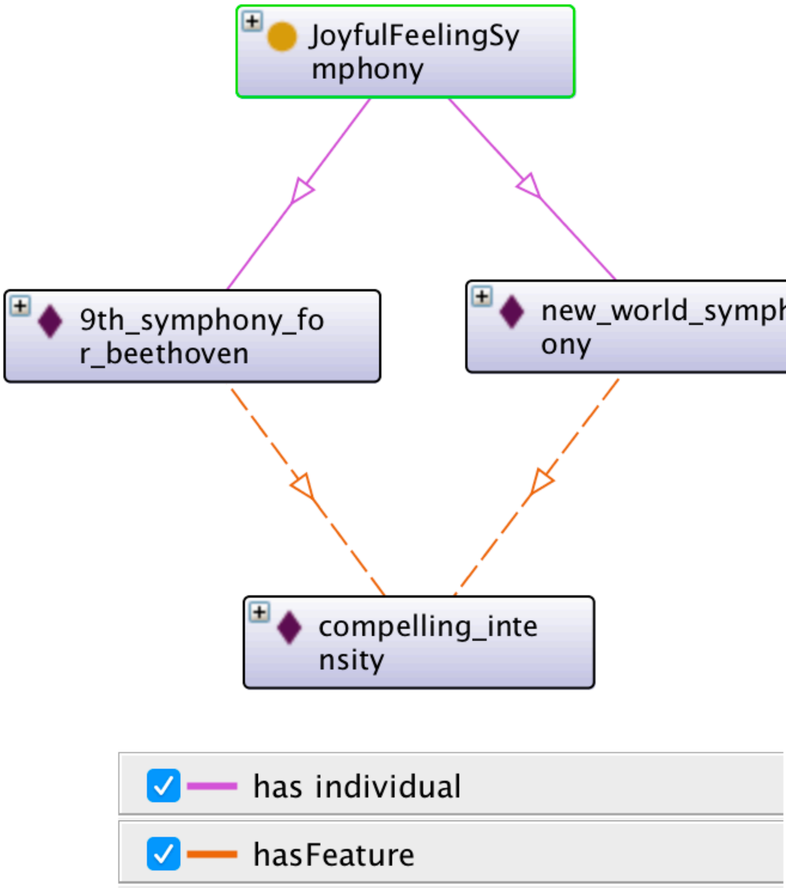


Figure 7-27 Level0 example

However, that is just the pure similarity between the items. Including the rating we calculate the similarity as:

$$PropertySimilarityLevel0 \times Level0Importance \times InstanceImportance \times rating + ClassSimilarityLevel0 \times Level0Importance \times ClassImportance \times rating$$

$$Sim = \frac{50}{100} \times \frac{2}{3} \times 1 \times 0.9 + \frac{75}{100} \times \frac{2}{3} \times \frac{5}{10} \times 0.9 = 0.525$$

7.7.8 Level 1 Instance Similarity

Section 7.7.2 shows an example of two instances *9th_symphony_for_beethoven* and *new_world_symphony* from level 1 because their values for the *Important Predicate composedBy*, which are *Beethoven* and *Dvorak*, share the same value, *romantic_era*, for the Important Predicate *fromMusicalEra*. Figure 7-28 shows a PropertySimilarity instance stating that *fromMusicalEra* has 40% similarity value. Applying the Similarity Equation, described in section 7.7.4, the system calculates the similarity as:

$$Sim = PropertySimilarityLevel1 \times Level1Importance \times InstanceImportance$$

$$Sim = \frac{40}{100} \times \frac{1}{3} \times 1 \approx 0.13$$

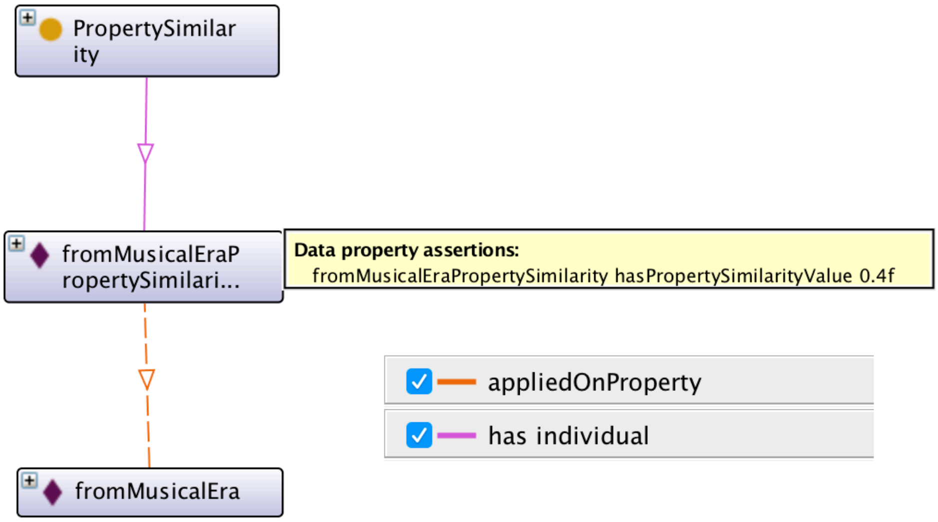


Figure 7-28 PropertySimilarity example 2

However, that is just the pure similarity between the items. Including the rating we calculate the similarity as:

$$Sim = PropertySimilarityLevel1 \times Level1Importance \times InstanceImportance \times rating$$

$$Sim = \frac{40}{100} \times \frac{1}{3} \times 1 \times 0.9 = 0.12$$

7.7.9 Level 1 Class Similarity

Figure 7-29 shows a rs:*ClassSimilarity* instance states that *Pianist* type has 40% similarity value. Applying the Similarity Equation, described in 7.7.4, the system calculate the similarity as:

$$Sim = ClassSimilarityLevel1 \times Level1Importance \times ClassImportance$$

$$Sim = 0.4 \times \frac{1}{3} \times 0.5 \approx 0.067$$

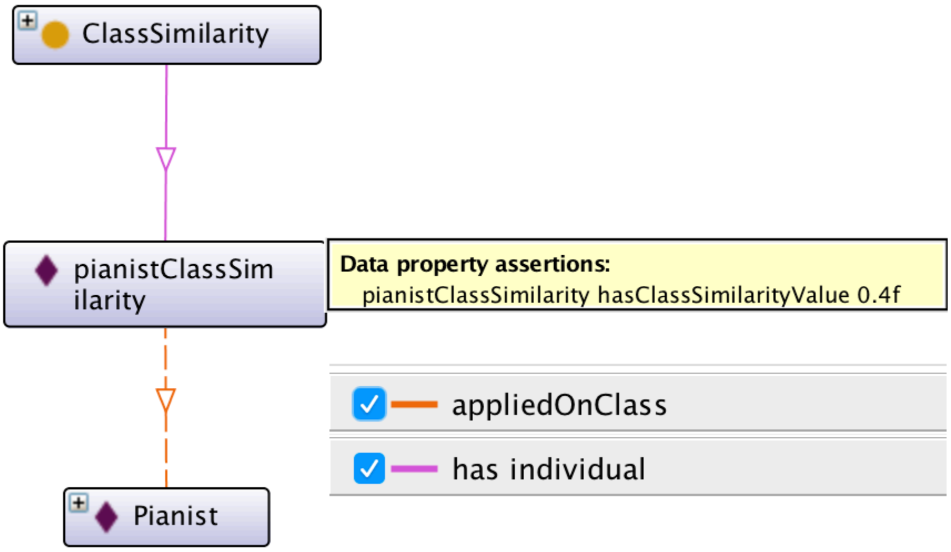


Figure 7-29 ClassSimilarity example 2

However, that is just the pure similarity between the items. Including the rating we calculate the similarity as:

$$Sim = ClassSimilarityLevel1 \times Level1Importance \times ClassImportance \times rating$$

$$Sim = 0.4 \times \frac{1}{3} \times 0.5 \times 0.9 = 0.06$$

7.7.10 Level 1 Both Instance and Class Similarity

Figure 7-30 is the result of combing the both examples provided in section 7.7.2, applying the Similarity Equation, described in section 7.7.4 the similarity between *9th_symphony_for_beethoven* and *new_world_symphony* is calculated as:

$$Sim = PropertySimilarityLevel1 \times Level1Importance \times InstanceImportance) + (ClassSimilarityLevel1 \times Level1Importance \times ClassImportance)$$

$$0.4 \times \frac{1}{3} \times 1 + 0.4 \times \frac{1}{3} \times 0.5 = 0.2$$

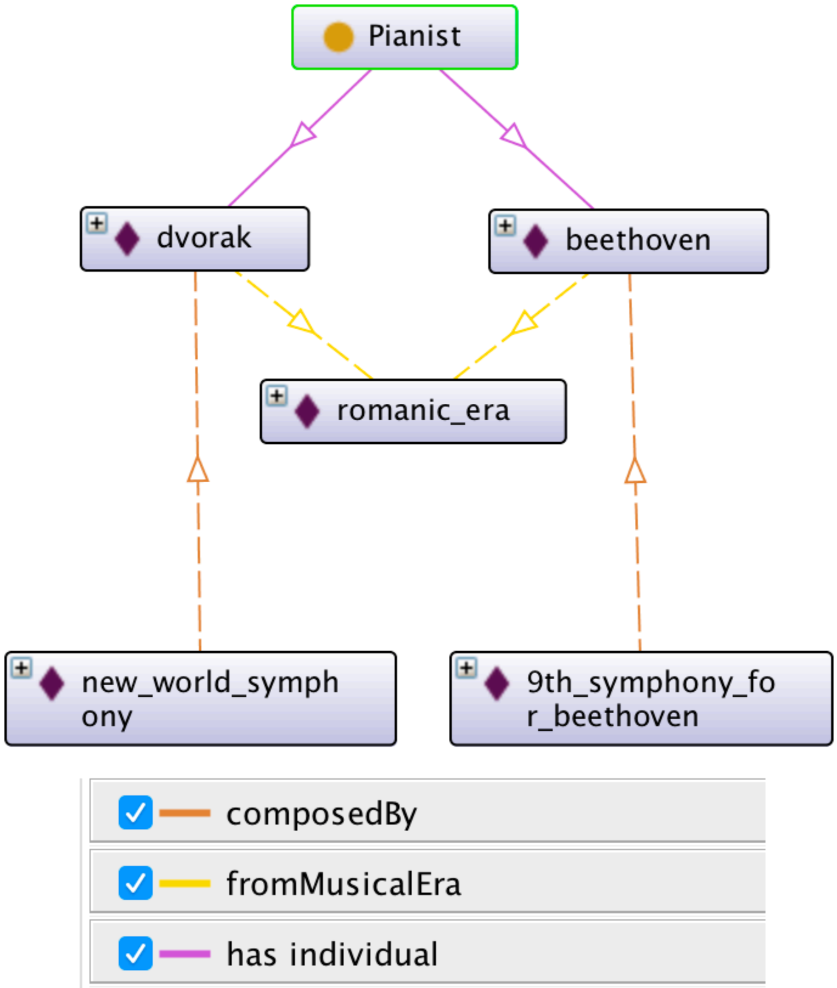


Figure 7-30 Level1 example

However, that is just the pure similarity between the items. Including the rating we calculate the similarity as:

$$Sim = PropertySimilarityLevel1 \times Level1Importance \times InstanceImportance \times rating) + (ClassSimilarityLevel1 \times Level1Importance \times ClassImportance \times rating)$$

$$0.4 \times \frac{1}{3} \times 1 \times 0.9 + 0.4 \times \frac{1}{3} \times 0.5 \times 0.9 = 0.18$$

7.7.11 Levels 0 and 1

Figure 7-31 is the result of combining the examples provided in sections 7.7.1 and 7.7.2 and the similarity values are the same as stated in sections 7.6.2, 7.6.3, 7.7.8 and 7.7.9.

Applying the Similarity Equation, described in section 7.7.4, the system calculates the similarity between the *9th_symphony_for_beethoven* and *the new_world_symphony* as:

$$Sim = (PropertySimilarityLevel1 \times Level1Importance \times InstanceImportance) + (ClassSimilarityLevel1 \times Level1Importance \times ClassImportance) + (PropertySimilarityLevel0 \times Level0Importance \times InstanceImportance) + (ClassSimilarityLevel0 \times Level0Importance \times ClassImportance)$$

$$Sim = \left(\frac{40}{100} \times \frac{1}{3} \times 1\right) + \left(\frac{40}{100} \times \frac{1}{3} \times 0.5\right) + \left(\frac{50}{100} \times \frac{2}{3} \times 1\right) + \left(\frac{75}{100} \times \frac{2}{3} \times 0.5\right) \approx 0.78$$

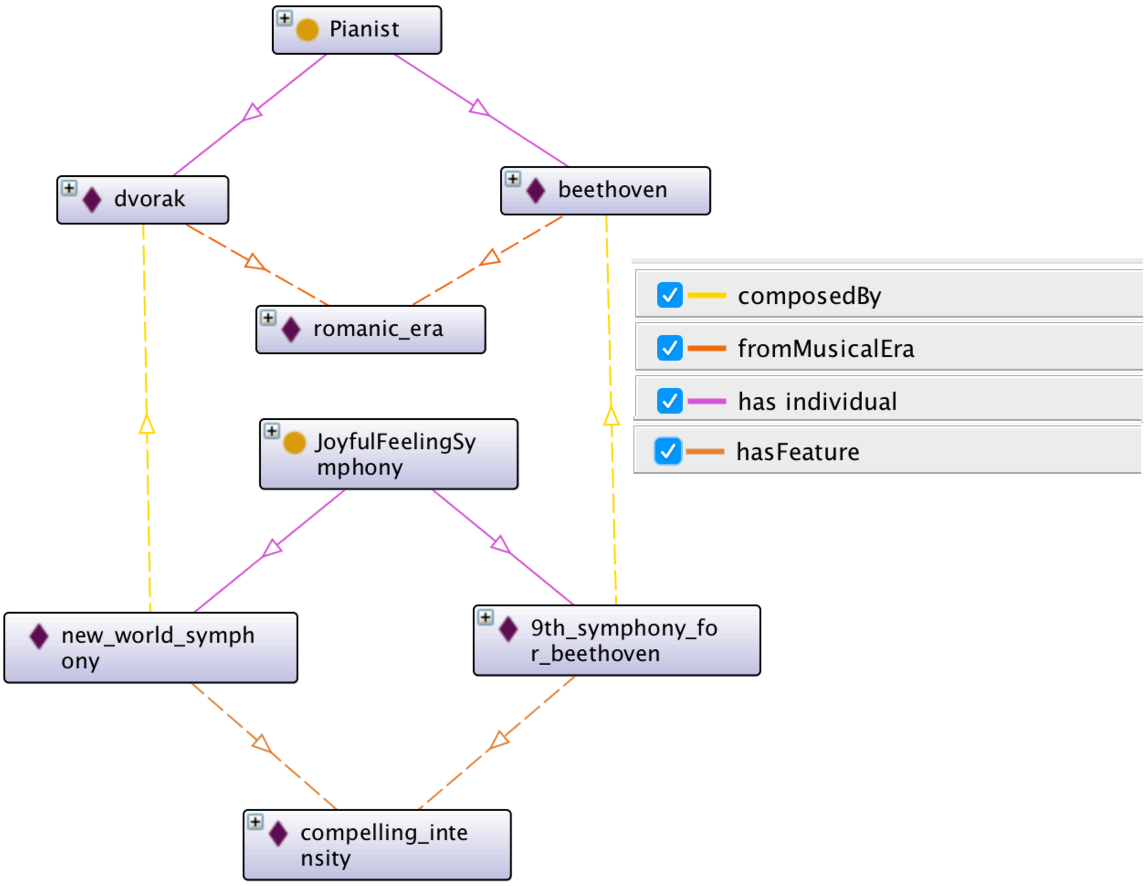


Figure 7-31 Level0 and Level1 example

However, that is just the pure similarity between the items. Including the rating we calculate the similarity as:

$$Sim = PureSimilarity \times rating) = 0.705$$

7.7.12 Level 0 and 1 more than on instance

Figure 7-32 and Figure 7-33 are two extensions to the example shown in Figure 7-31. In the same we calculated the similarity between *9th_symphony_for_beethoven* and *new_world_symphony*, we can find that the similarity between *9th_symphony_for_beethoven* and both the *5th_symphony_for_beethoven* and *nabucco_overture* are 0.12 and 0.48 respectively.

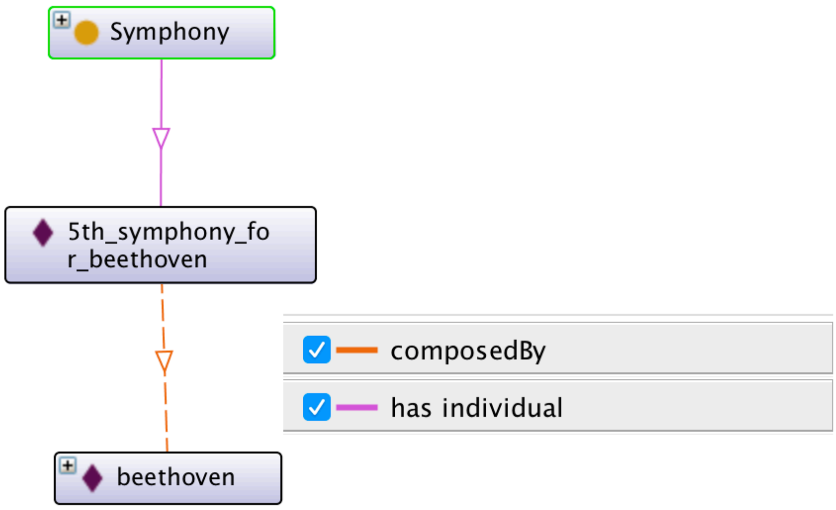


Figure 7-32 Level 0 and Level 1 Extension1

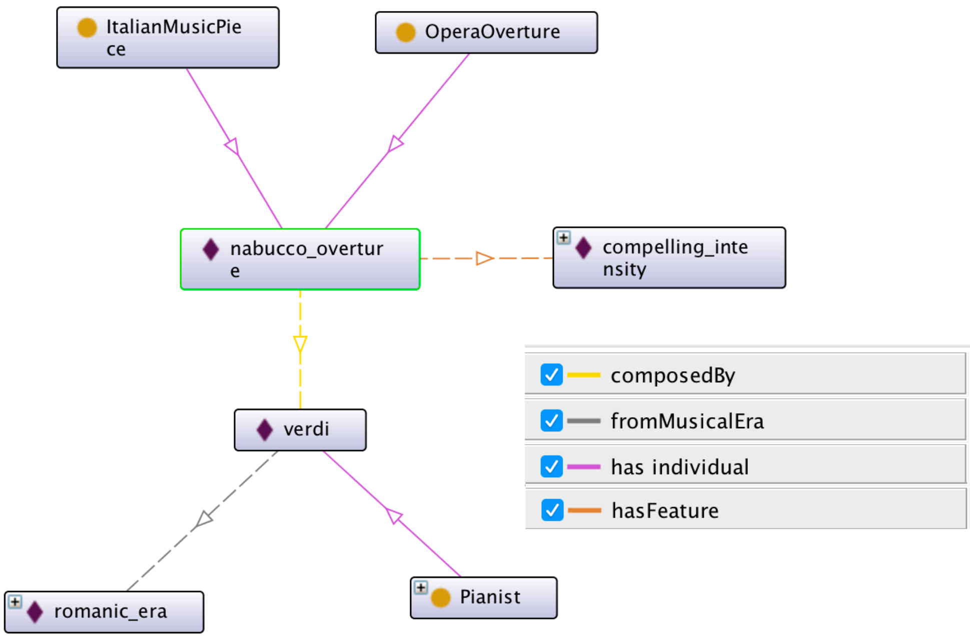


Figure 7-33 Level 0 and Level1 Extension2

As a result, the system will suggest *new_world_symphony*, then *nabucco_overture,* then *5th_symphony_for_beethoven* as shown in Table 7-1

Table 7-1 Galileo Galilei Recommendation

Recommended Item	Similarity	Reason
new_world_symphony	0.705	it shares the same class, which is JoyfulFeelingSymphony, with 9th_symphony_for_beethoven ,and it shares compelling_intensity for predicate hasFeature with 9th_symphony_for_beethoven ,and Both of beethoven and dvorak share romanic_era for fromMusicalEra ,and both of beethoven and dvorak are from type Pianist
nabucco_overture	0.48	it shares compelling_intensity for predicate hasFeature with 9th_symphony_for_beethoven ,and Both of beethoven and verdi share romanic_era for fromMusicalEra ,and both of beethoven and verdi are from type Pianist
5th_symphony_for_beethoven	0.12	it shares beethoven for predicate composedBy with 9th_symphony_for_beethoven

Figure 7-34 shows a new user *Max Black*, and his rating to the *9th_symphony_for_beethoven* is 0.9. His rating's value is the same as Galileo Galilei, so the system will suggest the same items as Galileo Galilei with the same similarities' values.

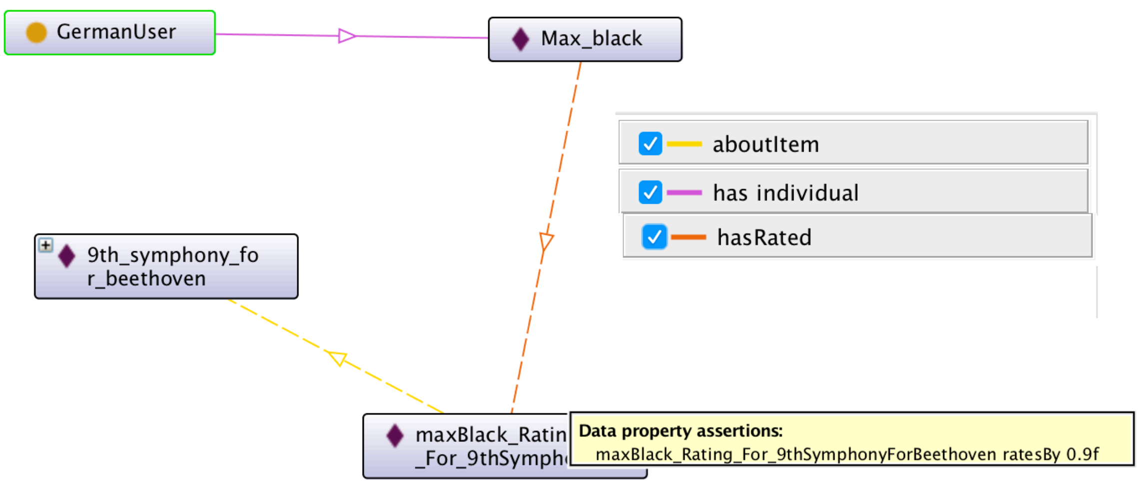


Figure 7-34 Max Black rating for the 9th_symphony_for_beethoven

7.7.13 Similarity with One User Context

We will suppose that *Galileo Galilei* is from *ItalianUser* class that has a user context as illustrated in section 7.6.5. Also, we will suppose that *nabucco_overture* is from *ItalianMusicPiece* class shown in the same section. After applying User Context, the user *Galileo Galilei* conforms to *ItalianUser* class, but the *User Context* instance does not have any value for *hasWeightIfContextMatched* so we will use the default value, which equals to 2, but can be configured differentially. Applying the Similarity Equation illustrated in section 7.7.4, the similarities can be calculated as the following:

$$LevelSimilarities \times\ UserContextWeight$$

Table 7-2 shows $SingleUserContextWeight$ value for *Galileo Galilei*.

Table 7-2 UserContext Weight for Galileo Galilei

Item	*User Context weight*	Reason
nabucco_overture	*2*	nabucco_overture is an instance of ItalianMusicPiece class and *Galileo Galilei* is an instance of ItalianUser class. However, italianMusicPieceItalianUserContext instance doesn't have a value for *hasWeightIfContextMatched* so the system uses the default value, which is 2.
new_world_symphony	*1*	new_world_symphony is not an instance of ItalianMusicPiece. Thus, the system uses the default value, which is 1[1].
5th_symphony_for_beethoven	*1*	5th_symphony_for_beethoven is not an instance of ItalianMusicPiece. Thus, the system uses the value of default value, which is 1.

The final similarities are illustrated in Table 7-3

[1] The system stores the default value for user context in a variable named defaultNoUserContext, the value should be static, though in the demo it can be changed but just for testing purposes.

Table 7-3 Galileo Galilei's similarities with one UserContext

Item	LevelSimilarities	UserContextWeight	Final Similarity
nabucco_overture	0.48	2	0.96
new_world_symphony	0.705	1	0.705
5th_symphony_for_beethoven	0.12	1	0.12

Now *nabucco_overture* became the first suggestion while it was the second suggestion for Galileo Galilei according to pure level similarities illustrated in Table 7-1.

Now consider a user *Max Black* who does not conform to *ItalianUser* class, but the User Context instance does not have any value for *hasWeightIfContextDoesNotMatch* so we will use the default value, which equals to 0.5, but can be configured differentially.

Table 7-4 shows UserContext weights for *Max Black.*

Table 7-4 UserContext Weight for Max Black

Item	User Context weight	Reason
nabucco_overture	0.5	Nabucco_overture is from type ItalianMusicPiece but Max Black is not from ItalianUser type. However, italianMusicPieceItalianUserContext instances does not have value for hasWeightIfContextDoesNotMatch. Thus, the system will use the default value, which is 0.5
new_world_symphony	1	new_world_symphony is not an instance of ItalianMusicPiece. Thus, the system uses the default value, which is 1.
5th_symphony_for_beethoven	1	5th_symphony_for_beethoven is not an instance of ItalianMusicPiece. Thus, the system uses the value of default value, which is 1.

The final similarities are illustrated in Table 7-5

Table 7-5 Max Black's similarities with one UserContext

Item	LevelSimilarities	UserContextWeight	Final Similaritiy

nabucco_overture	0.48	0.5	0.24
new_world_symphony	0.705	1	0.705
5th_symphony_for_beethoven	0.12	1	0.12

7.7.14 Similarity with many User contexts

Figure 7-35 shows another instance of UserContext class, *germanSymphonyGermanUserContext*, and which links the *GermanUser* class with *GermanSymphony* class. The instance has 0.2 as the value of hasWeightIfContextDoesNotMatch predicate, and 4 as the value of hasWeightIfContextMatched predicate. We will suppose that *5th_symphony_beethoven* is from type *GermanSymphony*.

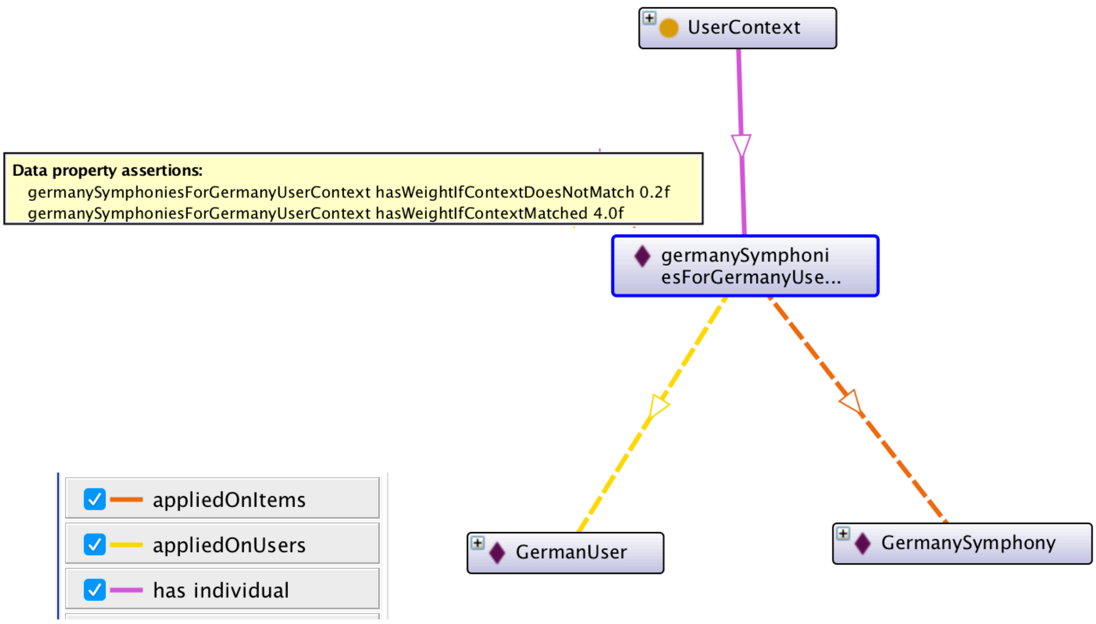


Figure 7-35 German Symphony UserContext

To calculate the similarities, the system needs to know the $UserContextWeight$ value as illustrated in the Similarity Equation 7.7.4, which its value in this case becomes the sum of the two *UserContext* instances weights the system has, as the following:

$$UserContextWeight = italianMusicPieceItalianUserContext\ value + germanSymphonyGermanUserContext\ value$$

The *italianMusicPieceItalianUserContext* values for *Galileo Galilei* and *Max Black* are already explained in Table 7-2 and Table 7-4 respectively, while *germanSymphonyGermanUserContext* values for Galileo Galilei and Max Black are described in Table 7-6 and Table 7-7 respectively.

Table 7-6 GermanSymphony User Context weights for Galileo Galilei

Item	UserContextWeight	Reason
nabucco_overture	1	*Nabucco_overture* is not from *GermanSymphony* class. Thus, the system uses the default value, which is 1
new_world_symphony	1	new_world_symphony is not from GermanSymphony class. Thus, the system uses the default value, which is 1
5th_symphony_for_beethoven	0.2	*5th_symphony_for_beethoven* is from *GermanSymphony* class but the user *Galileo Galilei* is not from *GermanUser* class. Thus, the system uses the value of *hasWeightIfContextDoesNotMatch* predicate, which is 0.2

Table 7-7 GermanySymphony User Context weights for Max Black

Item	UserContext Weight	Reason
nabucco_overture	1	*Nabucco_overture* is not from *GermanSymphony* class. Thus, the system uses default value, which is 1.
new_world_symphony	1	*new_world_symphony* is not from *GermanSymphony* class. Thus, the system uses the default value, which is 1
5th_symphony_for_beethoven	4	*5th_symphony_for_beethoven* is from *GermanSymphony* class and *Max Black* is from *GermanUser* class. Thus, the system uses the value of *hasWeightIfContextMatched*, which is 4

Figure 7-36 GermanSymphony user context weights for Max Black

The final similarities for *Galileo Galilei and Max Black* with many *UserContexts* are illustrated Table 7-8 and Table 7-9 in respectively.

Table 7-8 Galileo Galilei's similarities with many UserContexts

Item	LevelSimilarities	UserContextWeight	Final Similarity
nabucco_overture	0.48	2 + 1	1.44
new_world_symphony	0.705	1 + 1	1.41
5th_symphony_for_beethoven	0.12	1 + 0.2	0.144

Table 7-9 Max Black's similarities with many UserContexts

Item	LevelSimilarities	UserContextWeight	Final Similarity
new_world_symphony	0.705	1 + 1	1.41
nabucco_overture	0.48	0.5 + 1	0.72
5th_symphony_for_beethoven	0.12	1 + 4	0.6

We can notice that *5th_symphony_for_beethoven* now has a very low similarity value for *Galileo Galilei*, while it has a significantly larger value for *Max Black*.

7.7.15 Similarity with one Temporal Context

Section 7.6.6 shows how to create *TemporalContext* instance, *symphonyFestival2006*. It does not have any value for *hasWeightIfContextMatched* and *hasWeightIfContextDoesNotMatch* predicates. Thus, the system will use the default values. Assuming that the time of generating the recommendation is June, 2016. Thus, it is inside the range specified by *canBeRecommendedFrom* and *canBeRecommendedUntil* values.

The Similarity Equation, discussed in section 7.7.4, states that the final Similarity with Temporal Context is calculated as:

$$Sim(A, B) = LevelSimilarities \times TemporalContextWeight$$

Table 7-10 shows the *TemporalContextWeight* values for *Any User*[1].

Table 7-10 Temporal Context weights for any user

Item	TemporalContext Weight	Reason
new_world_symphony	1	*new_world_symphony* is not an instance of *GermanSymphony* class. Thus, the system uses the default value, which is 1

[1] The TemporalContextWeight values are the same for all users since it depends on the items not the users. In contract of the UserContextWeight values, which depend on both items and users.

nabucco_overture	1	*nabucco_overture* is not an instance of *GermanySymphony* class. Thus, the system uses the default value, which is 1
5^{th}_symphony_for_beethoven	2	*5th_symphony_for_beethoven* is an instance of *GermanySymphony* class, and the time of generating the recommendation is inside the range. However, the *TemporalContext* instance does not have a value for *hasWeightIfContextMatched* predicate. Thus, the system uses the default value for conforming to the context, which is 2

The final similarities for *Galileo Galilei* with one *TemporalContext* is illustrated in Table 7-11

Table 7-11 Final similarities for Galileo Galilei with one Temporal Context

Item	LevelSimilarities	TemporalContextWeight	Final Similarity
new_world_symphony	0.705	1	0.705
nabucco_overture	0.48	1	0.48
5th_symphony_for_beethoven	0.12	2	0.24

Notice that the weight of the *5th_symphony_for_beethoven* becomes higher. We will edit the *symphonyFestival2016* instance, illustrated in section 7.6.6 by adding 5 as the value of *hasWeightIfContextMatched* predicate, as illustrated in Figure 7-37

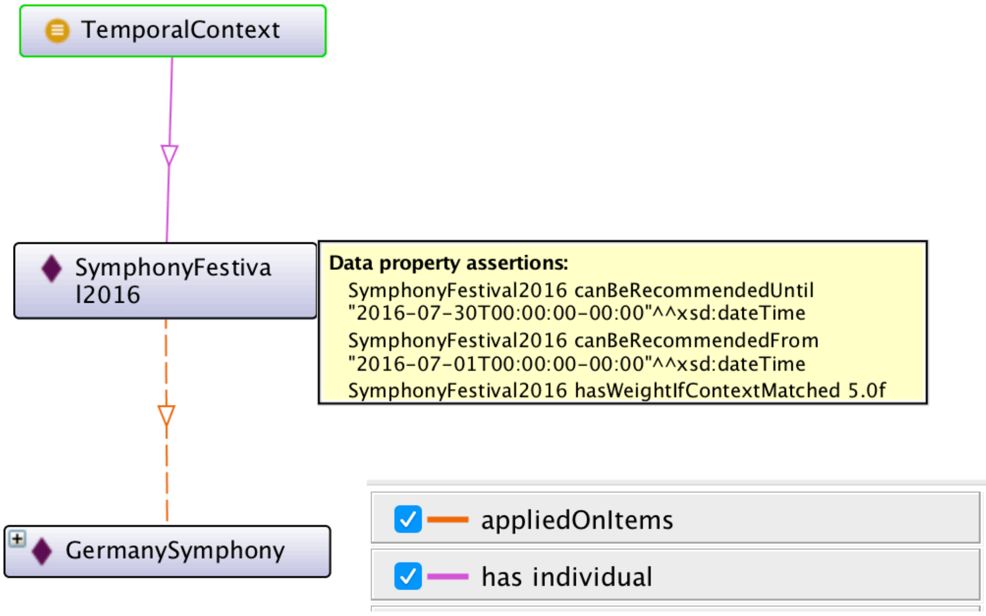


Figure 7-37 TemporalContext example with weight value

Now the *TemporalContextWeight* for all users regarding *5th_symphony_for_beethoven* becomes 5 instead of 2. Thus, the final similarities for both Galileo Galilei and Max Black are illustrated in Table 7-12

Table 7-12 Similarities after adding weight to the TemporalContext

Item	LevelSimilarities	TemporalContextWeight	Final Similarity
new_world_symphony	0.705	1	0.705
5th_symphony_for_beethoven	0.12	5	0.6
nabucco_overture	0.48	1	0.48

Comparing the new results with the results of pure level similarities, illustrated in Table 7-1, we can see that the *5th_symphony_for_beethoven* becomes the second recommended item instead of the third.

7.7.16 Similarity with many Temporal Contexts

Figure 7-37 illustrates another *TemporalContext* instance, operaWeek2016, that is applied on *OperalOverture* class with July 10^{th}, 2016 as the value of *canBeRecommendedFrom* predicate and July 17^{th}, 2016 as the value of *canBeRecommendedUntil* predicate, and 3 as the value of *hasWeightIfContextMatched* predicate.

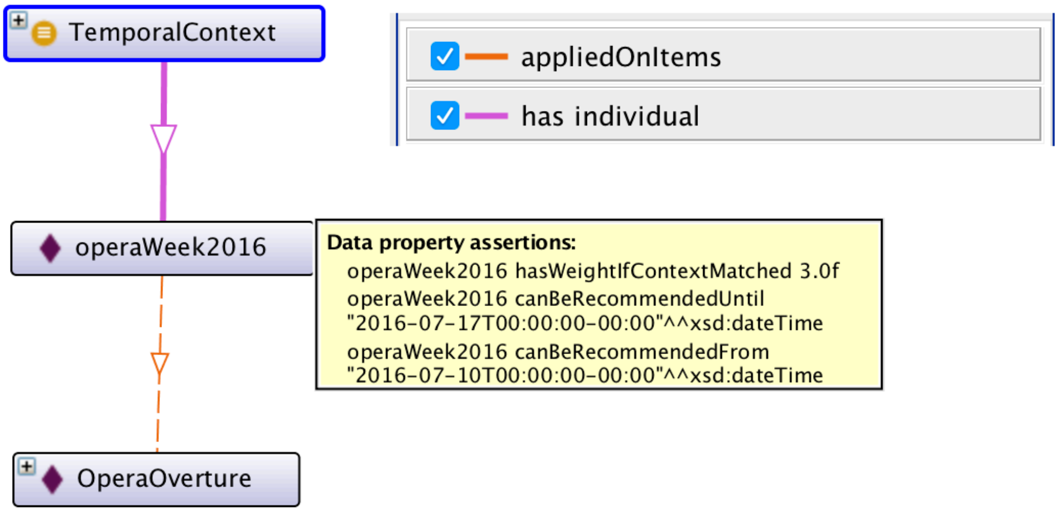


Figure 7-38 TemporalContext extension example

Assuming that the time of generating the recommendation is within the range specified by *canBeRecommendedFrom* and *canBeRecommendedUntil* values, the TemporalContextWeight regarding *operaWeek2016* context for all the users are illustrated in Table 7-13.

Table 7-13 TemporalContext weights regarding operaWeek2016 TemporalContext instance

Item	TemporalContextWeight	Reason
new_world_symphony	1	*new_world_symphony* is not an instance of *OperaOverture* class. Thus, the system uses the default value, which is 1
5th_symphony_for_beethoven	1	*5th_symphony_for_beethoven* is not an instance of *OperaOverture* class. Thus, the system uses the default value, which is 1
nabucco_overture	3	*nabucco_overture* is an instance of *OperaOverture* class, and the time of generating the recommendation is within

		the range of the *TemporalContext* instance operaWeek2016. Thus, the system uses the value of the *hasWeightIfContextMatched*, which is equal to 3.

Combining the weights from Table 7-13 and Table 7-12, the system is able to calculate the final weights for all TemporalContext instances, as illustrated in Table 7-14

Table 7-14 TemporalContextWeights for many TemporalContexts

Item	TemporalContextWeight for germanFestival2016	TemporalContextWeight for operaWeek2016	TemporalContextWeight
new_world_symphony	1	1	1 + 1 = 2
5th_symphony_for_beethoven	5	1	5 + 1 = 6
nabucco_overture	1	3	1 + 3 = 4

Applying the Similarity Equation, described in section 7.7.4, by joining the TemporalContextWeights, calculated in Table 7-14, with the pure level similarities for *Galileo Galilei*, calculated Table 7-1, the final similarities for both *Galileo Galilei* and *Max Black* [1]is illustrated in Table 7-15

Table 7-15 Recommendations with many TemporalContexts

Item	LevelSimilarities	TemporalContextWeight	Final Similarities
nabucco_overture	0.48	4	1.92
new_world_symphony	0.705	2	1.41
5th_symphony_for_beethoven	0.12	6	0.72

Comparing the recommendation without *TemporalContext*, illustrated in Table 7-1, with the recommendations with TemporalContext, illustrated in Table 7-15, we can see that *nabucco_overture* became first recommendations while it was second.

[1] The same values here for both users because both of them has exactly the same pure similarities since they've both ranked the same item with the same value, and TemporalContextWeight is not related to the users but to the items.

7.7.17 Similarity with UserContext and TemporalContext

Combining the pure similarity for *Galileo Galilei*, illustrated in Table 7-1, with the UserContextWeight, illustrated in Table 7-8, with the TemporalContextWeight, illustrated in Table 7-14, the final similarities can be calculated as in Table 7-16.

Table 7-16 Recommendations for Galileo Galilei with UserContext and TemporalContext

Item	LevelSimilarities	UserContextWeight	TemporalContextWeight	Final Similarities
nabucco_overture	0.48	3	4	5.66
new_world_symphony	0.705	2	2	2.82
5th_symphony_for_beethoven	0.12	1.2	6	0.864

Combining the pure similarity for *Max Black* with UserContextWeight, illustrated in Table 7-9, with the TemporalContextWeight, illustrated in Table 7-14, the final similarities can be calculated as in Table 7-17

Table 7-17 Recommendations for Max Black with UserContext and TemporalContext

Item	LevelSimilarities	UserContextWeight	TemporalContextWeight	Final Similarities
5th_symphony_for_beethoven	0.12	5	6	3.6
nabucco_overture	0.48	1.5	4	2.88
new_world_symphony	0.705	2	2	2.82

We can see that *Max Black* and *Galileo Galilei* have completely different sorting of the recommending items; the first recommended item for *Max Black* is the *5th_symphony_for_beethoven*, white it is the last for *Galileo Galilei*, and *nabucco_overture* is the first recommended item for *Galileo Galilei*, while it is the second for *Max Black*. Plus, *new_world_symphony* was the first recommended items for both of them when using just the pure similarities, while now it is not the first item for any of them.

7.7.18 Similarity with one Boosting

Section 7.6.8 shows an instance of *rs:Boosting* class, *operaOvertureBoosting*, with 2 as the value of *boostedBy* predicate. Table 7-1 shows the recommendations for *Galileo*

Galilei with Level 0 and Level 1. Applying the Similarity Equation, illustrated in Equation 22, we need to know the Boosting weight for each item. Table 7-18 shows these weights.

Table 7-18 Boosting weight for OperaOverture

Item	Boosting weight	Reason
new_world_symphony	1	*new_world_symphony* is not an instance of *OperaOverture* class. Thus, the system uses the default value, which is 1
nabucco_overture	2	*nabucco_overture* is an instance of OperaOverture class. Thus, the system uses the value of the *boostedBy* predicate, which is 2
5th_symphony_for_beethoven	1	*5th_symphony_for_beethoven* is not an instance of *OperaOverture* class. Thus, the system uses the default value, which is 1

Applying the Similarity Equation, the final similarities are illustrated in Table 7-19

Table 7-19 Recommendations with one Boosting instance

Item	Level Similarities	Boosting Weight	
nabucco_overture	0.48	2	0.96
new_world_symphony	0.705	1	0.705
5th_symphony_for_beethoven	0.12	1	0.12

Comparing the results with and without *Boosting*, we can see that *nabucco_overture* became the first recommended items while it was the second.

7.7.19 Similarity with UserContext, TemporalContext, and Boosting

Combining the examples illustrated in section 7.7.11 , 7.7.14, 7.7.16, and 7.7.18, and applying the Similarity Equation, we can find the final similarities as illustrated in

Item	LevelSimilarities	UserContext weight	TemporalContext weight	Boosting weight	Final Similarities
nabucco_overture	0.48	3	4	2	11.52
new_world_symphony	0.705	2	2	1	2.82
5th_symphony_for_beeth	0.12	1.2	6	1	7.2

oven					

7.7.20 Countable

Let us assume that the active user, *Michelangelo*, has the ratings shown in Table 7-20.

Table 7-20 Michelangelo's ratings

Item	Ratings value
requiem_sequentia	0.9
Le_nozze_di_Figaro	1.0
slavonic_dances_No_1	0.8

The *countableComposer* instance, illustrated in section 7.6.7, states that the system should recommend items that have the same composers as those items that the active user has liked. Analyzing the ratings of *Michelangelo*, the system fins out that *Michelangelo* has liked [1] two items composed by *Mozart*, and one item composed by *Antonin Dvorak*. As a result, the system should suggest other items composed by these two composers. However, in order to rank the items, the system uses a *Collaborative Filtering* approach in which it depends on other users' ratings to rank the suggested items. Let's suppose that the system has two other users, *Albert Einstein* and *Leonardo Da Vinci*, who have both rated *Slavonic Dances Number 3* and *Serenade Eine Kleine Nachtmusik* as shown in Table 7-21

Table 7-21 Albert Einstein and Leonardo Da Vinci ratings values

	Slavonic Dances No 3	Serenade Eine Kleine Nachtmusik
Albert Einstein	0.9	1
Leonardo Da Vinci	0.8	0.8

The ranking will depend simply on the average of ratings.

Notes:

1. The system does not check the items for all the composers that the active user has liked, but it does that just for a specific number of composers, and that number is configurable. For instances, if that number is 5, the system first extracts the most 5 composers that the active user has liked symphonies composed by them, then the system uses the ratings to suggest the items.

[1] In other words, the active user has rated them high enough so the system can consider that the active user likes the items.

2. The system does not suggest all the items rated by the other users, but it suggests just a specific number of items, and that number is configurable. For instance, if that number is 10, so the system suggests just the best 10 symphonies according to other users' ratings.
3. The system does not suggest all the items for the other users' ratings, but it suggest the items that their ratings' average is more than a specific value, this value is configuration in a variable called *averageRatingValue*.

The items that the system extracts/recommends from the Countable feature do not replace the main recommendations coming from combining Pure Similarities, UserContext and TemporalContext, but it provides another stream of recommendations. Plus, there is a stream of recommended items for each CountableConfiguration instance.

Assuming that the domain experts want to choose the most 3 composers the users have liked, and the *averageRatingValue* is 0.6.

The user *Michelangelo* has liked two items composed by Mozart and one item composed by Antonio Dvorak. The system checks other items, composed by those two composers, that other users have rated them. The average ratings of *serenade_eine_kleine_nachtmusik* is 0.9 because according to Table 7-21, Albert Einstein rated it by 1, and Leonardo da Vinci rated it by 0.8. Table 7-22 shows the final recommendations for Michelangelo regarding the composedBy countable instance.

Table 7-22 Recommendations for Michelangelo regarding composedBy countable instance

Item	Average rating	More from
serenade_eine_kleine_nachtmusik	0.9	Mozart
slavonic_dances_No_3	0.85	Antonio Dvorak

However, if the value of *averageRatingValue* is 0.9 instead of 0.6, the system will only suggest serenade_eine_kleine_nachtmusik.

We will extend the ratings of *Albert Einstein* and *Leonardo Da Vinci*, illustrated in Table 7-21, by adding more items and more ratings for them as illustrated in Table 7-23

Table 7-23 Albert Einstein and Leonardo Da Vinci extension ratings values

	symphon_in_D_Major_K_385	Piano_sonata_no_12
Albert Einstein	0.9	0.9
Leonardo Da Vinci	0.9	1

The new recommendations for Michelangelo are listed in Table 7-24

Table 7-24 New recommendations for Michelangelo regarding composedBy countable instance

Item	Average Rating	More From
Piano_sonata_no_12	0.95	Mozart
serenade_eine_kleine_nachtmusik	0.9	Mozart
symphon_in_D_Major_K_385	0.9	Mozart

As we can see, the three recommended items are composed by Mozart. However, if wanted to generated 4 items instead of three, the fourth item would be *slavonic_dances_No_3*, because its average rating is 0.85 as listed in Table 7-25

Table 7-25 New recommendations or Michelangelo regarding composedBy countable instance 2

Item	Average Rating	More From
Piano_sonata_no_12	0.95	Mozart
serenade_eine_kleine_nachtmusik	0.9	Mozart
symphon_in_D_Major_K_385	0.9	Mozart
slavonic_dances_No_3	0.85	Antonio Dvorak

However, to avoid the problem of having all the items for the same composer, and thus having more diverse recommendations, which is usually more required by users, the Semantic Recommender can alter the current approach and use the one in the next section.

7.7.21 Countable with specific number of items for each value

All the recommendations shown in Table 7-24 are from a specific composer, which is Mozart. We tried to solve that problem in Table 7-25 by increasing the number of recommending items. However, in a real life, we do not know the exact number of recommending items that we should generate to get items from diverse composers. To solve that problem, the system will suggest a specific number of items for each composer. So the system will select a specific umber of composers, and for each of them, it will select a specific number of items according to other users' ratings. In this case, there is a new variable, which is the *numberOfItemsForEachCountableValue* that specifies the number of recommending items for each composer. Supposed that its value is 2, the recommending items for *Michelangelo* are listed in Table 7-26

Table 7-26 New recommendations for Michelangelo regarding composedBy countable instance 3

Item	Average Rating	More From

Piano_sonata_no_12	0.95	Mozart
serenade_eine_kleine_nachtmusik	0.9	Mozart
slavonic_dances_No_3	0.85	Antonio Dvorak

7.8 Discussion

7.8.1 Possible advantages

- We do not need to search the whole information repository to find similar items to a specific one as CB approach does.
- The accuracy is supposed to be better than the other approaches because domain experts are the ones who get to define the criteria for item similarities, and specify their similarity values. While CB approach depends mainly on automatic discovery for the relationships between items' features.
- This approach suits the best for business measures from evaluation point of view because with Boosting, Countable and UserContext services, we can recommend diverse items, which is normally an important requirement for any business.
- This approach is supposed to scale good even if the Ontology is big because we are just working on two levels. In other words, two arcs far in the RDF graph. However, as we will see in the experiment, we need a good SPARQL server. Nevertheless, it does not require pre calculations as the other approaches do.
- This approach is easy to integrate in already productive systems; it works with any domain because all we need to do is creating a domain Ontology and the Joined Ontology. This is a hug advantages over both the CB approach that needs to analyze the content for each domain, and over CF approach that needs to apply complex-and-expensive mathematical algorithms to find the similarities.
- This approach exceeds over CF approach for the very-important problem, which is the Cold Start problem, because this approach just needs to know at least one items that the user likes, and it will suggest similar items semantically, while CF approaches struggle in a complex mathematical algorithms to solve this problem. The experiments with TIMWE data, illustrated in section 9.1.4, prove that.
- It is easy to recommend more items; we just need to add a third level, which is Level 3. In other words, modify the SPARQL query by adding one more UNION clause. While in CF approaches we would need to re calculates the model again because when the model is being calculated, there is a factor K states how many

similar items we should calculate for a specific item, this factor in CF approach is extremely critical to improve the performance of operations. Otherwise, the similarities will be saved for all the items, and that is not possible in a real business application [1].

- This approach excels in providing very accurate justifications. While in CF approach there are no justifications at all since there is no context from the first place. The users and the items are just numbers in a huge matrix. CB approach provides justifications but not so accurate because the attributes that are being used to analysis the items are mainly Meta data while in our approach the features of the items are being specified by domain experts using the domain Ontology.

7.8.2 Possible disadvantages

- This approach is from CB family. Thus, it does not take into considerations the collaborative features. However, it uses CF features in the Countable service.
- The Ontology has to be detailed enough, because if it is not, the system will generate poor quality recommendations.
- Until now defining the Important Predicates and the Important Classes are being manually. However, we could improve that by creating a small wizard that reads all the predicates and all the classes in the domain Ontology and provides an interface for the user to specify their similarity values.

[1] Imagine a store like Amazon saving the similarities between all its more-than-million products, and yet keep calculating over and over again every week or so.

Men create gods after their own image, not only with regard to their form, but with regard to their mode of life.

Aristotle

Don't kid yourself that you're going to live again after you're dead; you're not. Make the most of the one life you've got. Live it to the full.

Richard Dawkins

I contend that we are both atheists. I just believe in one fewer god than you do. When you understand why you dismiss all the other possible gods, you will understand why I dismiss yours

Stephen Henry Roberts

8 Implementation

The aims of this chapter are three-fold:

- First, describe the subcomponents of the Semantic Recommender and the way they work together.
- Second, describe the available recommendation services and the way they are being consumed/requested.
- Third, list the development problems, the shortages and the bugs of the used tools.

This chapter focuses on implementing Semantic Recommender, illustrated in Figure 7-1. The Recommender Ontology and the Joined Ontology were already described in details in chapter 7.

8.1 Semantic Recommender System

8.1.1 Architecture

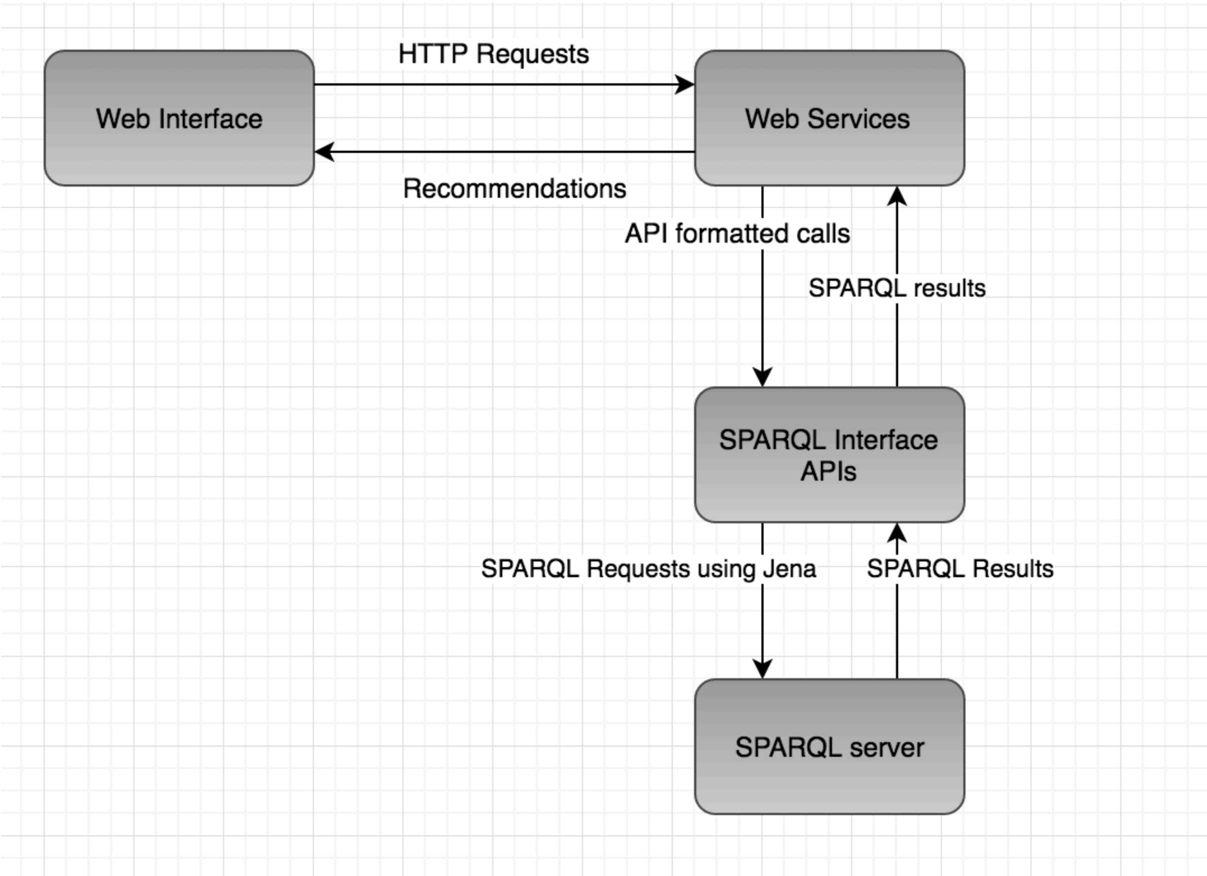


Figure 8-1 Semantic Recommender System Subcomponents

Figure 8-1 shows sub components of the Semantic Recommender System, which are:

1. SPARQL Server

Fuseki[1] is the selected SPARQL server; it's the component that implements SPARQL Protocol, receive SPARQL queries and execute them on the stored RDF graph.

2. SPARQL Interface API

It is a JAVA component that provides APIs to be consumed by the *Web Service* component. Each API does the following tasks:

1. Loading the correct SPARQL template.
2. Customizing it according to the request coming.
3. Prepare the template if the user has requested justification
4. Calling the SPARQL server with the generated SPARQL query
5. Returning the results to the *Web Service* component.

This component has a configuration file, *config.properties*, which contains:

- The SPARQL endpoint URL
- Domain Ontology prefix
- Domain Ontology URI
- Recommender Ontology prefix
- Recommender Ontology URI
- Joined Ontology prefix
- Joined Ontology URI

3. Web Services (Semantic Recommender Web Services)

It is a JAVA component that provides web services to be consumed by the Web Interface component, extracts the required values from the request, calls the corresponding API from the *SPARQL Interface API* component, receives the results, formats them, and generates a HTTP response. In general, this component simulates the interaction between the existing domain system with the Semantic Recommender.

4. Web Interface

It is a PHP web application that final users can use to test the proposed system. It provides many ways to ask for recommendations. It allows the users to customize their requests by selecting the users who they want to generate recommendations for. It also allows

[1] https://jena.apache.org/documentation/serving_data/

final users to provide their own values for the parameters being used in calculating the similarities, such as *Level1Importance* and *InstanceImportance*.

8.1.2 Class Diagram

Figure 8-2 shows class diagram for the Semantic Recommender Web Services component, while Figure 8-3 shows class diagram for Sparql Interface component.

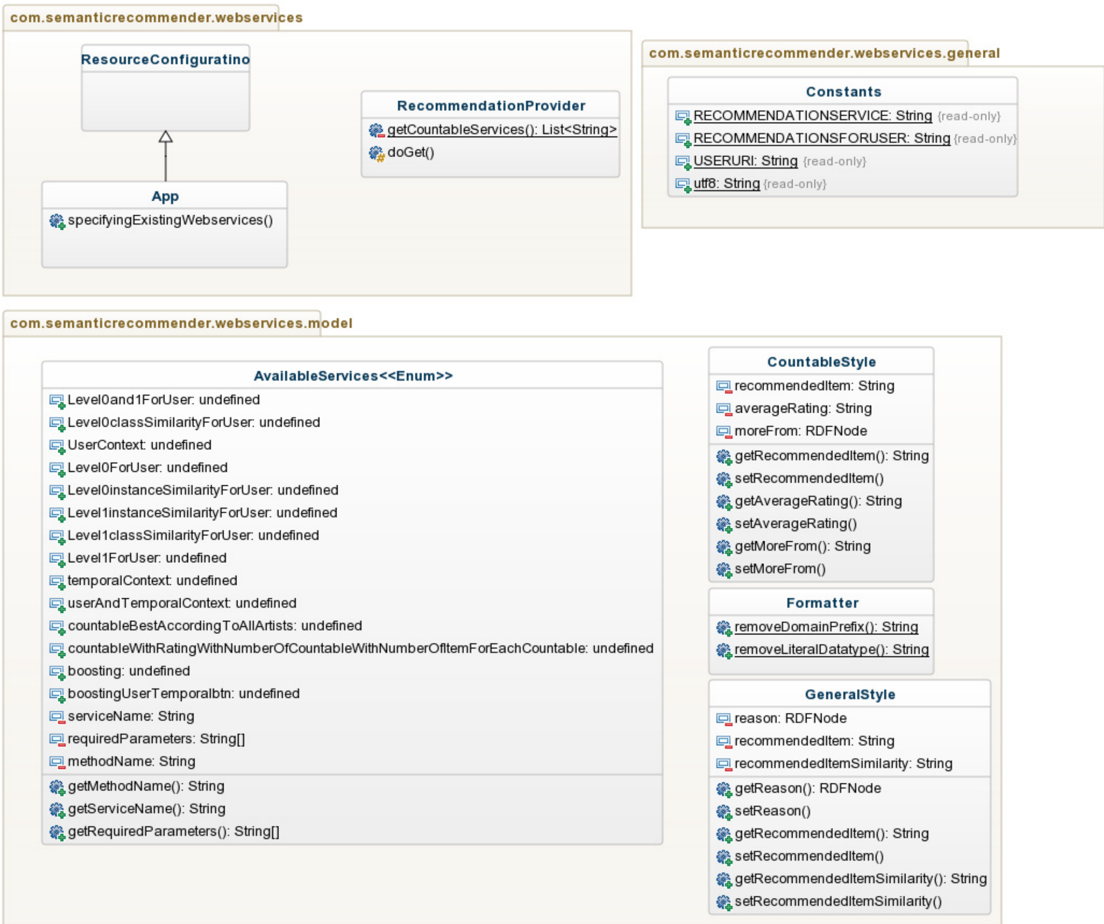


Figure 8-2 Semantic Recommender Web services Class Diagram

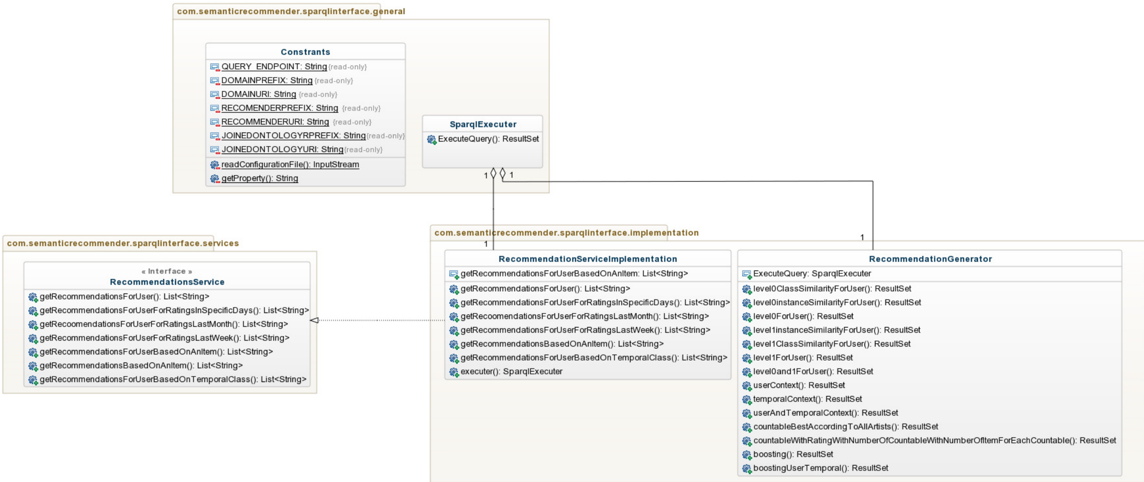


Figure 8-3 Sparql Interface Class Diagram

8.1.3 Sequence Diagrams

A sequence diagram for the first Use Case (Get Recommendations) assuming that the Domain System requests the level 0 instance recommendation service is illustrated in Figure 8-4 (To see the image in a big quality, please request this URL: https://www.mediafire.com/?np3sle13hciccjc)

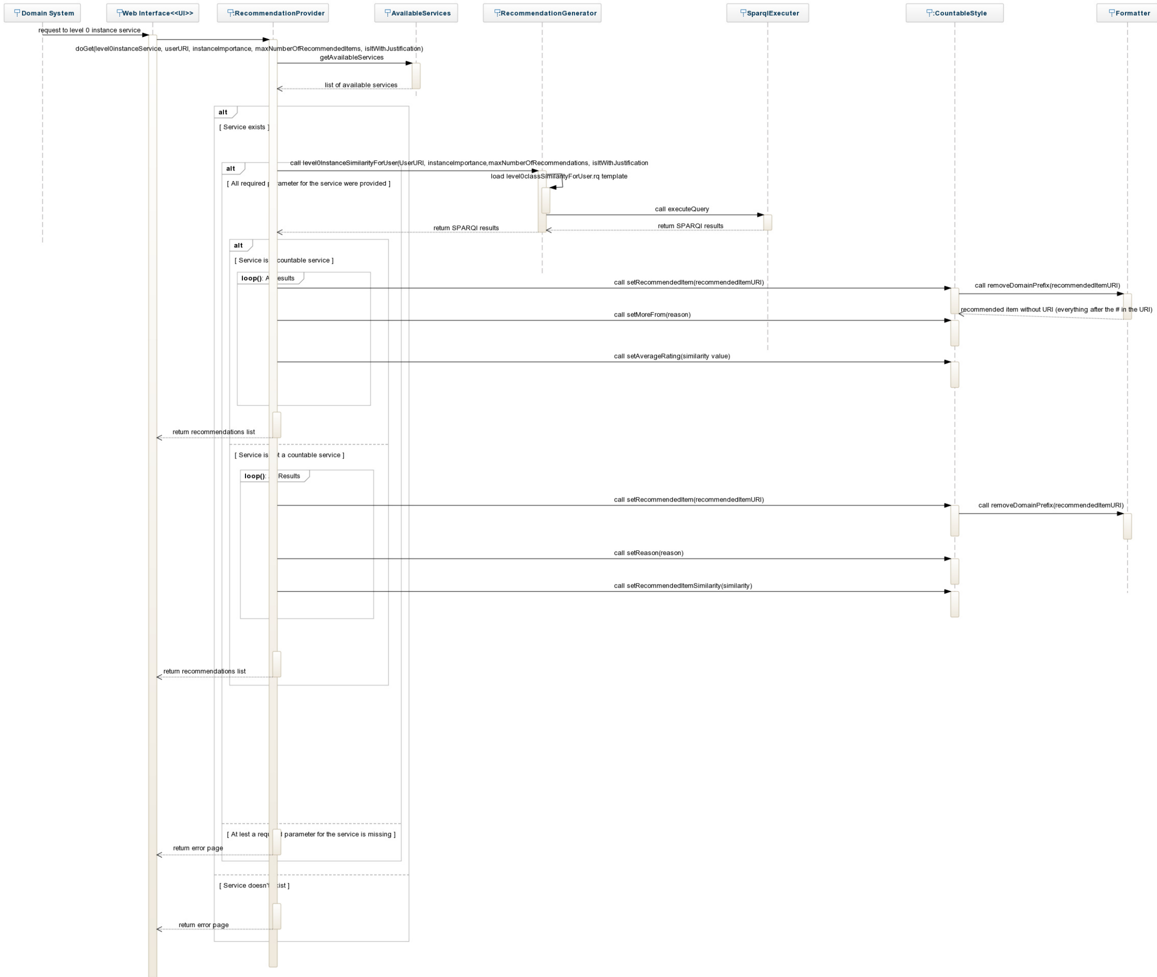


Figure 8-4 Sequence Diagram Level 1 Instance Service

8.2 Available Recommendation Services

There are many available services that the domain administrator can request. Each service is independent of the other. So, each one generates a stream of recommendations. Some of the services are: Level 0 Instance Recommendation Service, Level 1 Recommendation Service, and User Contexts Recommendation Service. The full list of services with the parameters for each services are described in the Annex 10.5.

8.3 Problems and shortages

Some versions of Fuseki do not recognize OWL files.

Some Ontology(ies) work with OWL Full, and that is not supported by Fuseki reasoned.

Most of the Ontology(ies) do not provide A-Box assertions like the Wine ontology described in section 9.3.2.

Sometimes we could find A-Box assertions but without the corresponding OWL Ontology, such as LinkedMDB [1].

We have tried to extract/infer the schema of a Music Ontology using its RDF dump, but all the SPARQL endpoints provided by on this website http://dbtune.org/ are not working.

I could find some Ontology(ies) with T-Box and A-Box but there is no ratings.

Some Ontology(ies) are not good for recommendations[2].

For some reasons, if you have the T-BOX assertions and the A-Box assertions of in different files, loading both of them into Protégé, protégé will make the object properties as annotations. So we would need to write properties manually for each instance when we were trying test ontologies such as the Book ontology.

DBpedia does not provide Schema, it just provides triples, and most of the useful data is classes, not properties, as we will see in the DBpedia music Ontology in section 9.3.3.

Fuseki TDB service crashes after uploading a large amount of data. In other words, we can upload a large amount of data, but when restarting Fuseki, Fuseki crashes, which means we have to upload the data again every time.

Bug in Jena version 3.0.0 when the query contains both group_contact and distinct; if the query is this:

$$select\ (group\_concat(distinct\ ?x)\ as\ ?y)\ (sum(distinct\ ?x)\ as\ ?z)\ \{\}$$

Jena 3.0.0 parse it to:

$$SELECT\ (GROUP\_CONCAT\ DISTINCT\ (?x)\ AS\ ?y)\ (SUM(DISTINCT\ ?x)\ AS\ ?z)\ WHERE\ \{\}$$

Which is illegal. We submitted a ticket and they corrected that but in version 3.1.0 in May 2016

While protégé supports OWL 2 reasoner, Fuseki does not; Fuseki's reasoner, which is Jena's reasoner, supports only OWL 1 reasoner. This was a big disadvantage because OWL classes defined by OWL datatype restrictions will not work with Fuseki. For instance, if we have a User subclass, MatureUser, that is begin defined as:

$$User\ and\ (hasAge\ some\ xsd{:}int[\geq\ 18\ \hat{}\hat{}xsd{:}int])$$

It will work on Protégé but not with Fuseki, and it is being stated explicitly on Jena's documentation "OWL2 vocabulary. NOTE: Jena does not provide OWL2 inference or OntModel support." [3].

[1] The RDF dump for LinkedMDB can be downloaded from http://www.cs.toronto.edu/~oktie/linkedmdb/

[2] Here are many examples http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library

[3] https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/vocabulary/OWL2.html

When importing an Ontology into another Ontology in Protégé, protégé does not generate visual triples for the imported Ontology. In other words, when we joined the Domain Ontology with the Recommender Ontology, we were not able to show visual representations of the triples, which is vitally important to present all the examples illustrated in chapter 7. We have reported a bug on February 2016, and an unofficial solution was provided on May 2016.

In protégé the object of the OWL Object property must be an instance, but in RDF, the object of any predicate can be a class or an instance. Thus, to put a class as the object of an OWL property in protégé, we would create an instance of the THING class that has the same URI as the needed class.

I have not failed. I have just found 10000 ways that won't work

Thomas Edison

9 Experiments

The aims of this chapter are four-fold:

- First, describe the TIMWE case study, create Ontology to model its data, and implement the full Ontology Populating process.
- Second: evaluate the proposed solution with TIMWE's case.
- Third: evaluate the proposed solution with MovieLens case.
- Fourth: evaluate the proposed solution with other domains such as books and music.

To test the proposed solution, we need to have a TBox and ABox for the business domain. Plus, we should have ratings for the users in order to know their preferences. In section 9.1, we present TIMWE case study in which we create a corresponding Ontology, and change its content format to RDF, then we evaluate the proposed solution. We do the same in section 9.2 with Movielens case. Finally in section 9.3, we test the Recommender System with many other domains including books, wine and music.

9.1 TIMWE Case Study

TIMWE is a global provider of mobile monetization solutions for mobile carriers, media groups, governments/NGOs, brands and end-customers, focusing on Mobile Marketing, Mobile Entertainment and Mobile Money. It provides a platform for content providers so they can publish their content whatever it is. Thus, TIMWE does not generate content itself, but it has content from many domains, and its system is built to allow the providers to add extra information to their content, such as content-type, category, description, creator … and so on. Recently, they have started asking the users for their opinions about the content.

TIMWE's data is stored in a database that contains around 200 tables. However, just some of them are beneficial to build an Ontology such as Language, Artist and Album, while the others are for technical purposes such as queues, web services, etc.

ER diagram can be found on this link https://www.mediafire.com/?7wo0gl7q0g77b5u It is a proprietary resource for TIMWE, not allowed to be copied or shared.

9.1.1 TIMWE Content

1. Content Types

TIMWE provides data in many types, which are:

- Polyones: Polyphonic ringtones, or "polys", can use up to 64 notes at once, far more than the single note monophonic can produce[1]. This results in a richer sound that can better match "real" music.
- Truetones: recordings of actual music. While polyphonic ringtones can only simulate music by combining preset sounds of instruments in a musical pattern, real-music ringtones (truetones or realtones) are the actual music. Unlike polyphonic ringtones, real-music ringtones can contain vocals
- Special Sounds: technically the same as Truetones, however, Special sounds are recordings of specific Sounds rather than actual songs.
- Ringbacktones: Audio that is heard by the originator of the call while is calling.
- Videos: Video clips that are downloaded to the mobile device in a short version of 10, 30 and 60 seconds.
- Full Videos: Video clips that are downloaded to the mobile device in a full version.
- Videotones: ringtones with real sounds and live images that play at the same time during an incoming call. In other words, Videotones are a mix of Truetones with video
- Monotones: it is not used anymore, it was just for legacy devices.
- Wallpapers: Static graphics used to represent the screen background of the mobile phone and formatted to fit the screen.
- Animations: animated graphics created in gif format.
- Screensavers: animated graphics used to represent the screen background of the mobile phone and formatted to fit the screen. The image is repeated in an indefinite loop. Screensavers are produced using animated gif formats.
- Games: applications that are specifically designed to play on mobile devices. There are several different technologies for downloadable games, including Java, BREW, Flash Lite, Windows Mobile, BlackBerry, Android, etc.
- Applications: programs for mobile devices with the exception of mobile games, which are classified under Games. Examples of applications are Slideshows, Widgets, Mobile maps, Chat, etc. Applications can be developed for Java (J2ME), Symbian, Windows Mobile, BlackBerry, Android, etc.
- Themes: A theme is a way to change the graphical experience for the user in just one setting.

[1] Monophonic, Polyphonic, and Homophonic are music textures. For a simple description, you may watch this video https://www.youtube.com/watch?v=Vg8vVZ0IUTA

- Text: The ‘text’ product is a messaging service that is sent via SMS or viewed on a web page. Examples of text services are ‘love tips’, ‘horoscope’, ‘weather news’.
- SMS Alerts

Each type of them has different data fields, as illustrated in Table 9-1, where *M* stands for mandatory and *O* stands for optional.

Table 9-1 TIMWE content type attributes

Content-type	PKEY	Title	Artist	Album	Short Descripti	Long Descripti	Genre	Price	PricePoint ID	Device Capabilit	ASset
Truetones	M	M	M	O	O	O	M	O	O	O	M
Special Sounds	M	M	M	O	O	O	M	O	O	O	M
Ringbacktones	M	M	M	O	O	O	M	O	O	O	M
Video	M	M	O	O	O	O	M	O	O	O	M
Full Video	M	M	M	O	O	O	M	O	O	O	M
Videotones	M	M	O	O	O	O	M	O	O	O	M
Polytones	M	M	O	O	O	O	M	O	O	O	M
Monotones	M	M	O	O	O	O	M	O	O	O	M

SMS Alerts	Text	Themes	Applications	Games	Screensavers	Animations	Wallpapers
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
O	O	O	O	O	O	O	O
O	O	O	O	O	O	O	O
M	M	M	M	M	O	O	O
O	O	O	O	O	O	O	O
M	M	M	M	M	M	M	M
O	O	O	O	O	O	O	O
O	O	O	O	O	O	O	O
O	O	M	M	M	O	O	O
O	O	M	M	M	M	M	M

2. Transaction

Each transaction contains many fields, some of them are [1]:

1. CUSTOMER_MSISDN
2. MEDIA_PKEY
3. TYPE_ID

[1] I didn't list the whole columns in each transaction because this is private information related to TIMWE Company. I listed just the columns that are related to my research.

3. *Users' Info*

TIMWE does not save any information for the users except the mobile number, because that is the only field required to buy the content.

4. *Ratings*

TIMWE has started to ask users to rate the content. The followings columns are what they store:

1. MEDIA_ID
2. APPUSER_ID
3. RATING_VALUE
4. RATING_DATE

5. *Limitation*

The rating feature in TIMWE is new, so until now they just have 254 ratings.

9.1.2 Data Modeling Component

The Data Modeling component, described previously in the general architecture section 7.2, contains two processes. For the *Ontology Creating Process*, we manually analyze TIMWE's database, and built a small Ontology to model its data. The Ontology contains 8 classes, 5 OWL object properties, and 1 OWL data property [1]. For the *Ontology Populating Process*, we transferred the relational data to RDF depending on the examples provided by W3C about how to map relational data to RDF [59]. We accessed the database and created RDF triples using D2RQ [2], which is a platform that provides read-only RDF graphs for relational database.

During the next sections, it is described how TIMWE Ontology's classes were populated [3].

1. *Language Class*

This class represents the language of the media. The data is taken from a relational database table, ENUM_LANG, which is described in Figure 9-1, and to transfer its content to RDF format, we consider each row as an instance that has a URI formatted as:

---

[1] We choose <u>*to*</u> as the prefix for TIMWE's Ontology.

[2] http://d2rq.org/

[3] This is a mix between Ontology Creating Process and Ontology Populating Process.

TIMWE Prefix + language + primary key value

If the primary key of a language is 29, the corresponding instance will have the following URI:

http://www.timwe.com/to#language29

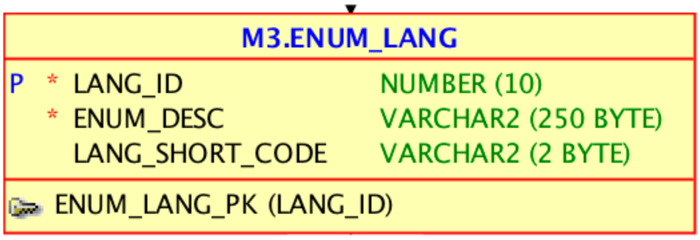


Figure 9-1 ENUM_LANG relational database table

Each *to:Language* instance has an OWL Data Property, *to:hasName*; its range is *xsd:string* type and its value is taken from the column *ENUM_DESC*. An example of an *to:Language* instance is shown in Figure 9-2

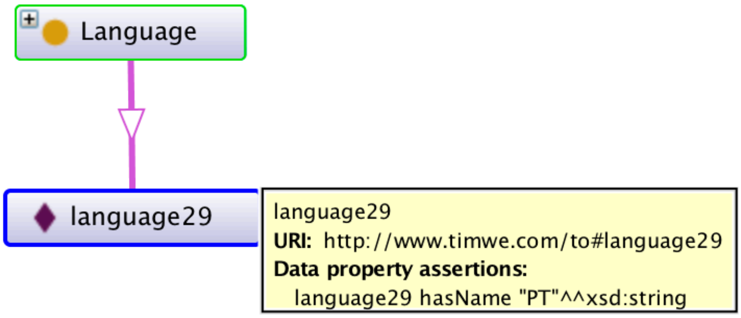


Figure 9-2 Language instance example

2. Genre Class

This class represents the genre of the media; its content is taken from joining two relational database tables, *GENRE* and *GENRELANG* that are described in Figure 9-3

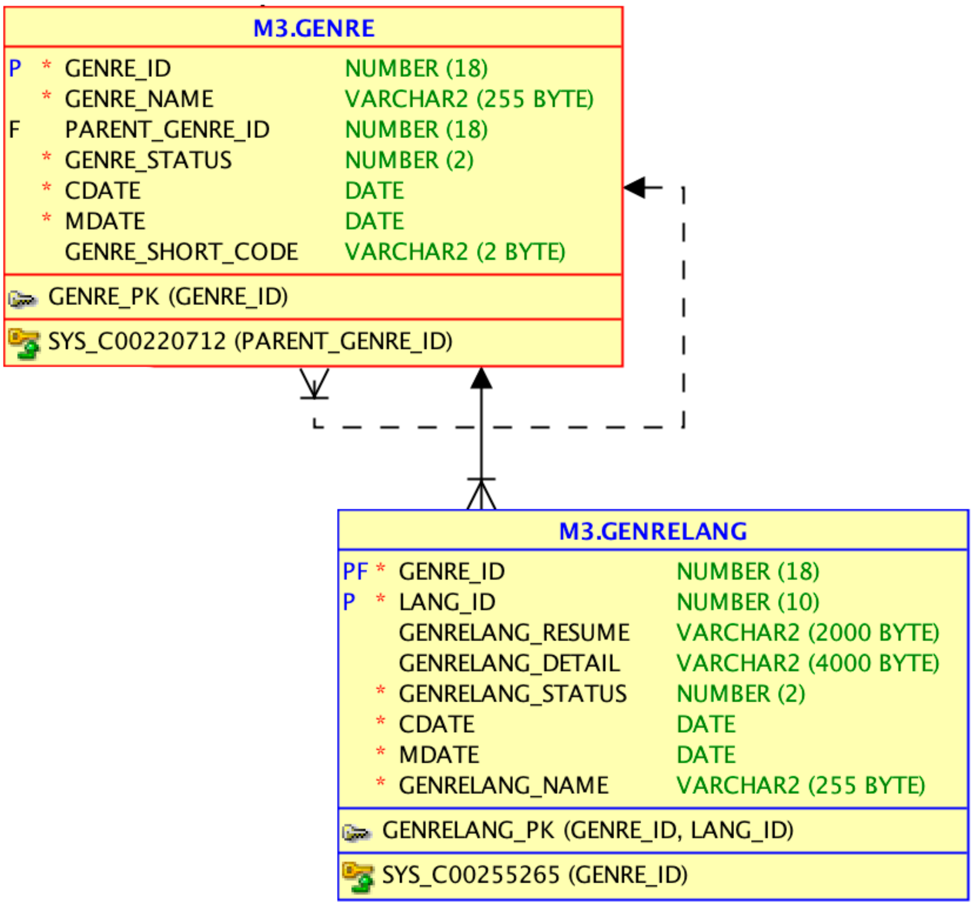


Figure 9-3 GENRE and GENRELANGE relational database tables

To transfer the content of these two tables into RDF format, we do the same as we have done with the Language table.

Each *to:Genre* instance could have one of more *to:hasLanguage* OWL Object Property, that links the *to:Language* instances to their *to:Genre* instances. To do so, the system examines the table *GENRELANG* and creates a *to:hasLanguage* property that links the *to:Genre* that its URI is taken from the *GENRE_ID* column with the *to:Language* instance that its URI is taken from *LANG_ID* column.

Each *to:Genre* instance could have *to:hasParentGenre* OWL Object Property, that links that instance to another to:Genre instance. To do so, the system examines the table *GENRE* and creates a *to:hasParentGenre* property that links the *to:Genre* that its URI is taken from *GENRE_ID* column with the *to:Genre* that its URI is taken from the *PARENT_GENRE_ID* column

An example of a *to:Genre* , its *to:Language* and its *to:hasParentGenre* is shown in Figure 9-4

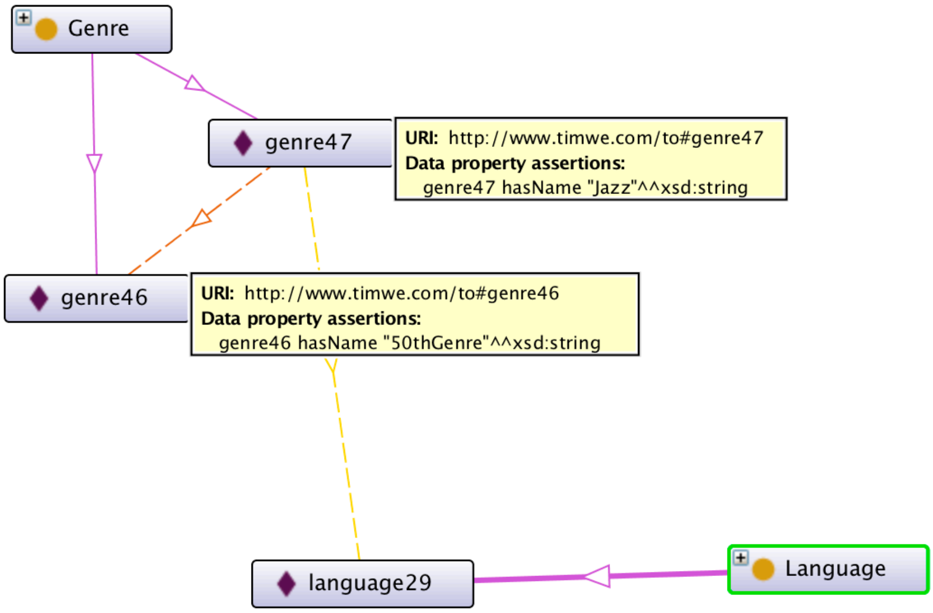
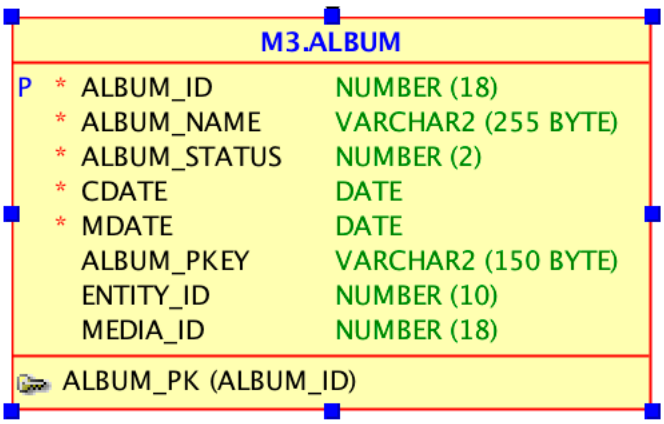


Figure 9-4 Genre instance example

3. *Album Class*

This class is to represent the album of the media. The data is taken from a relational database table, ALBUM, that's described in Figure 9-5



M3.ALBUM			
P	*	ALBUM_ID	NUMBER (18)
	*	ALBUM_NAME	VARCHAR2 (255 BYTE)
	*	ALBUM_STATUS	NUMBER (2)
	*	CDATE	DATE
	*	MDATE	DATE
		ALBUM_PKEY	VARCHAR2 (150 BYTE)
		ENTITY_ID	NUMBER (10)
		MEDIA_ID	NUMBER (18)
		ALBUM_PK (ALBUM_ID)	

Figure 9-5 ALBUM relational database table

To transfer the content of that table to RDF format, we do the same as we have done with Language class. Each *to:Album* instance has *to:hasName* OWL Data property.

. Figure 9-6 is an example

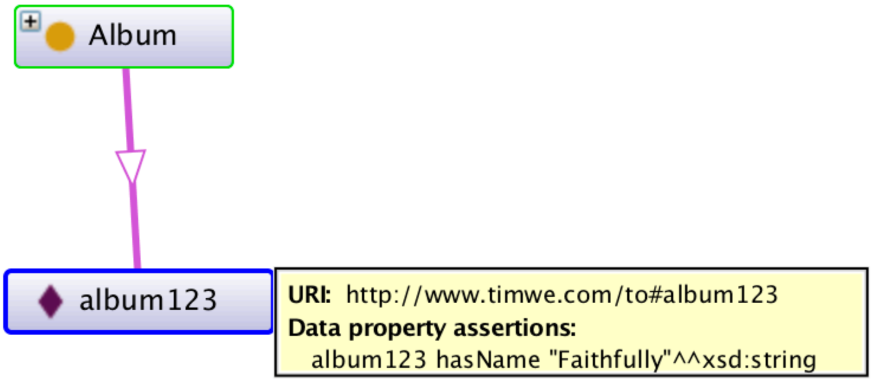


Figure 9-6 Album instance example

4. Artist Class

This class represents the artist of the media, mainly the singers. Its data is taken from two relational database tables, *ARTIST* and *ARTISTALBUM*, which are described in Figure 9-7

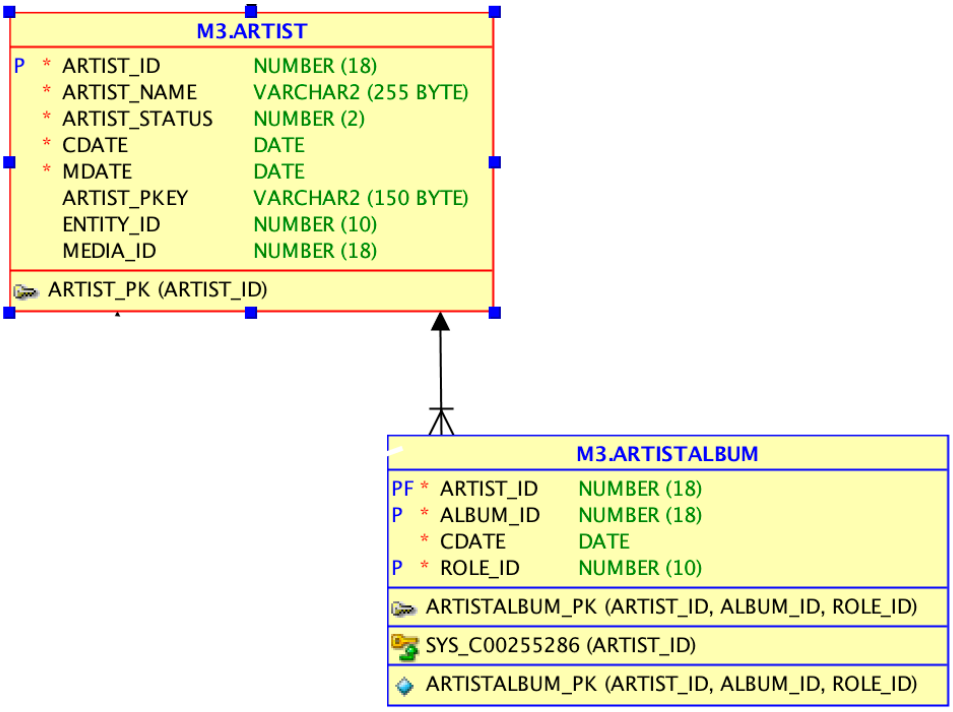


Figure 9-7 ARTIST and ARTISTALBUM relational database tables

To transfer the content of these two tables into RDF format, we do the same as we have done with *to:Language* and *to:Album* classes. Each *to:Artist* instance has *to:hasName* OWL Data Property, and could have one or more *to:hasAlbum* OWL Object Property, and one or more *to:hasGenre* OWL Object Property (coming from the relational database *ARTISTGENRE*, which is described in Figure 9-8).

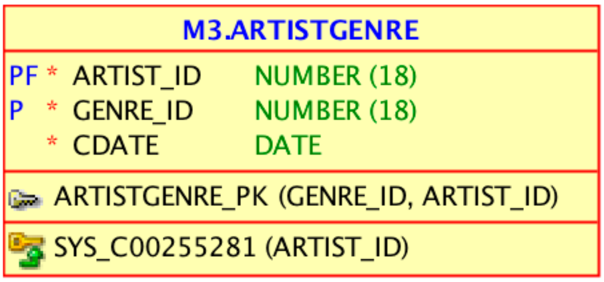


Figure 9-8 ARTISTGENRE relational database table

An example of a *to:Artist*, its *to:Album*, and its *to:Genre* is shown in Figure 9-9

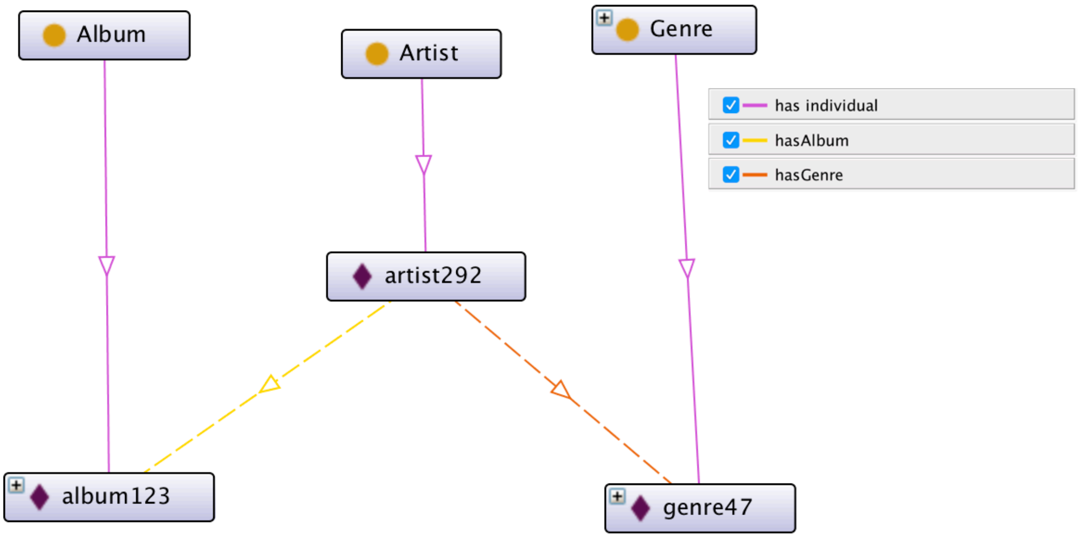


Figure 9-9 Artist instance example

5. Category Class

This class represents the category of the media; its data is taken from the relational database CAT [1]. To transfer the content of this table to RDF format, we do the same as before. Each *to:Category* instance has an *to:hasName* OWL Data Property. An example of an *to:Category* instance is shown in Figure 9-10

[1] I did not put the structure of that table since it is similar to ALBUM table.

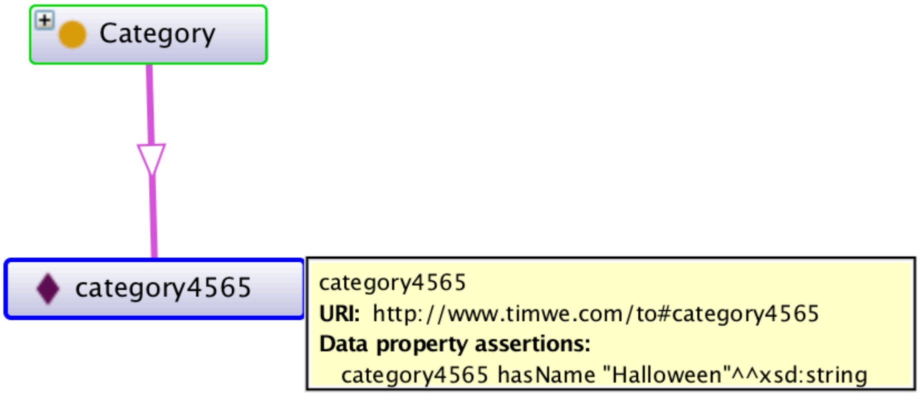
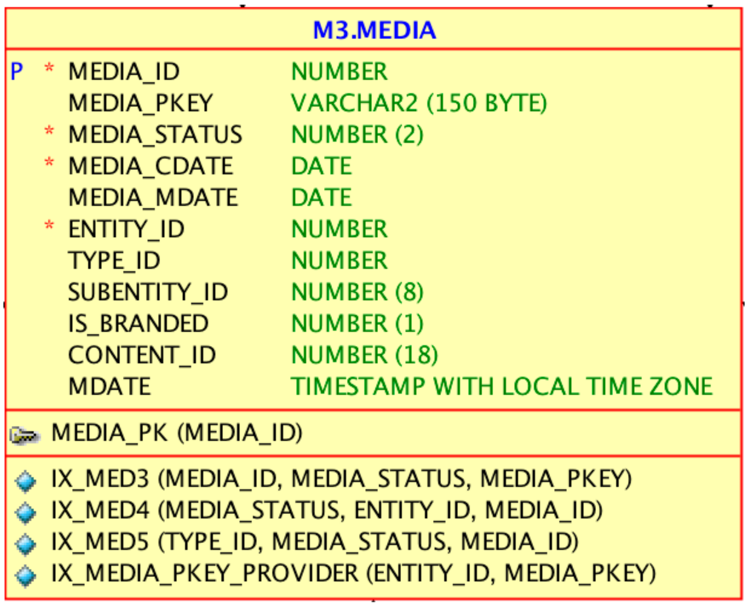


Figure 9-10 Category instance example

Note that the CAT class does have other attributes like CAT_CLASS, but we will not consider them because their value do not account to the similarities between items. The distinct categories classes' values are: Partner, Free, Premium, and Normal. As a result, knowing that two medias are free is not an indicator for the similarity between them.

6. Media Class

It represents the actual items that the system will recommend to the users. Its data is taken from a relational database table, *MEDIA* that is described in Figure 9-11



M3.MEDIA		
P *	MEDIA_ID	NUMBER
	MEDIA_PKEY	VARCHAR2 (150 BYTE)
*	MEDIA_STATUS	NUMBER (2)
*	MEDIA_CDATE	DATE
	MEDIA_MDATE	DATE
*	ENTITY_ID	NUMBER
	TYPE_ID	NUMBER
	SUBENTITY_ID	NUMBER (8)
	IS_BRANDED	NUMBER (1)
	CONTENT_ID	NUMBER (18)
	MDATE	TIMESTAMP WITH LOCAL TIME ZONE
	MEDIA_PK (MEDIA_ID)	
	IX_MED3 (MEDIA_ID, MEDIA_STATUS, MEDIA_PKEY)	
	IX_MED4 (MEDIA_STATUS, ENTITY_ID, MEDIA_ID)	
	IX_MED5 (TYPE_ID, MEDIA_STATUS, MEDIA_ID)	
	IX_MEDIA_PKEY_PROVIDER (ENTITY_ID, MEDIA_PKEY)	

Figure 9-11 MEDIA relational database table

To transfer the content of this table to RDF format, we do the same as before. Each *to:Media* instance could have one or more *to:hasArtist* OWL Object Properties (coming from the *MEDIAARTIST* table, which is described in Figure 9-12), and could have one or more *to:hasAlbum* OWL Object Property[1] (coming *MEDIAALBUM* table, which is described in Figure 9-13), and one or more *to:hasLanguage* OWL Object Property (coming from *MEDIALANG* table, which is described in Figure 9-14), one or more *to:hasGenre* OWL Object Properties (coming from *MEDIAGENRE* table, which is described in Figure 9-15), one or more *to:hasCategory* OWL Object Properties (coming from CATMEDIA table, which is described in Figure 9-16)

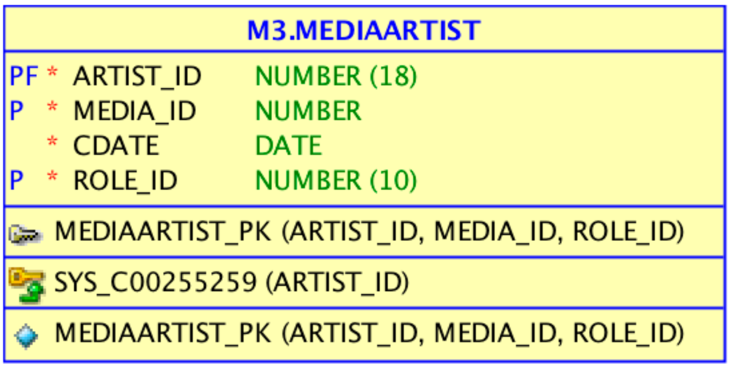


Figure 9-12 MEDIAARTIST relational database table

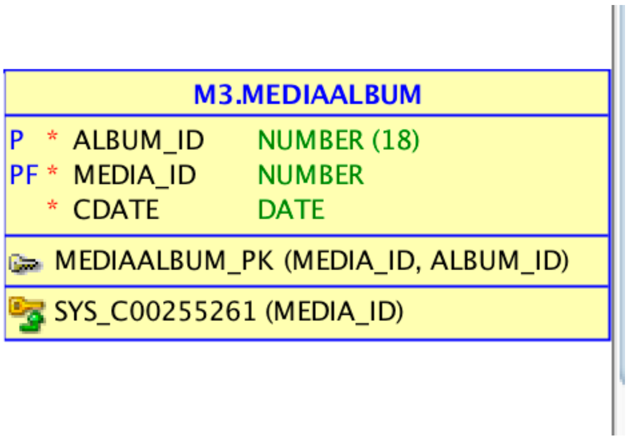


Figure 9-13 MEDIAALBUM relational database table

[1] Though in the current music each song belongs to one specific album, but there are also some songs, specially the old ones and the classical pieces, that belong to many albums.

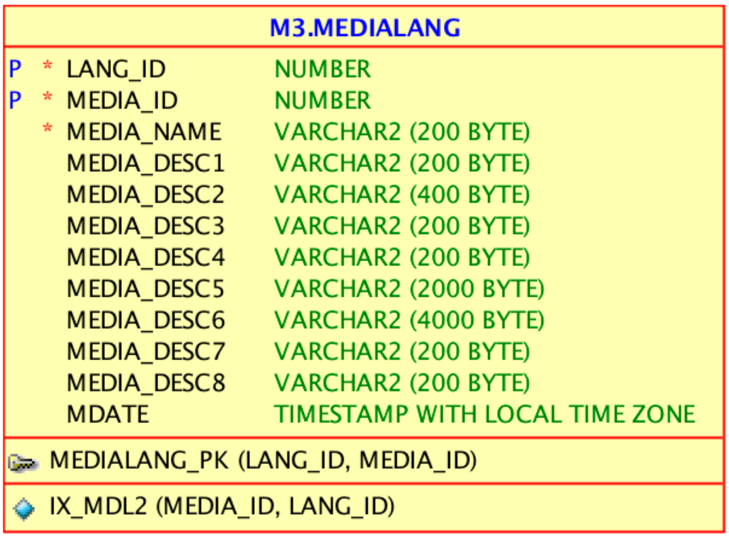


Figure 9-14 MEDIALANG relational database table

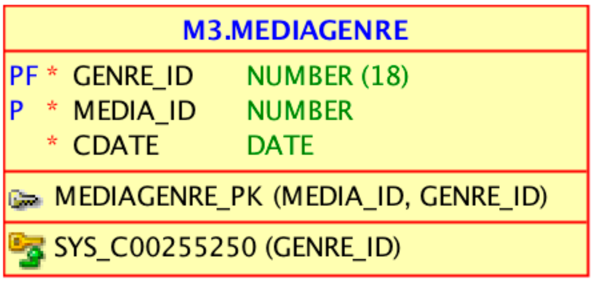


Figure 9-15 MEDIAGENRE relational database table

M3.CATMEDIA			
P	*	CAT_ID	NUMBER
P	*	MEDIA_ID	NUMBER
	*	CATMEDIA_POS	NUMBER (18)
	*	CATMEDIA_NAME	VARCHAR2 (200 BYTE)
		CATMEDIA_DESC1	VARCHAR2 (200 BYTE)
		CATMEDIA_DESC2	VARCHAR2 (400 BYTE)
		CATMEDIA_DESC3	VARCHAR2 (200 BYTE)
	*	CATMEDIA_OKEY	VARCHAR2 (100 BYTE)
	*	CATMEDIA_STATUS	NUMBER (10)
		CATMEDIA_DESC4	VARCHAR2 (200 BYTE)
		CATMEDIA_DESC5	VARCHAR2 (2000 BYTE)
		CATMEDIA_DESC6	VARCHAR2 (4000 BYTE)
		CATMEDIA_DESC7	VARCHAR2 (200 BYTE)
		CATMEDIA_DESC8	VARCHAR2 (200 BYTE)
	*	CATMEDIA_CDATE	DATE
		CATMEDIA_MDATE	DATE
F	*	BILLINGTYPE_ID	NUMBER (18)
	*	CATMEDIA_RANK	NUMBER (2)
		PRICE	NUMBER (10,4)
	*	DOWNLOADS	NUMBER
	*	VOTES	NUMBER
	*	RATING	NUMBER
		PRICEPOINT_ID	NUMBER (18)
		SALESTYPE_ONESHOT	NUMBER (1)
		SALESTYPE_SUBSCRIPTION	NUMBER (1)
		BILLING_PLAN_ID	NUMBER (18)
CATMEDIA_PK (CAT_ID, MEDIA_ID)			
SYS_C00255134 (BILLINGTYPE_ID)			
CAT_MEDIA_NAME_IDX (CATMEDIA_NAME)			
IX_CATMEDIA3 (MEDIA_ID, CAT_ID, CATMEDIA_STATUS)			
IX_CATMEDIA_MEDIA (MEDIA_ID)			
CAT_MEDIA_DESC1_IDX (CATMEDIA_DESC1)			
CAT_MEDIA_OKEY_IDX (CATMEDIA_OKEY)			

Figure 9-16 CATMEDIA relational database table

An example of *to:Media* instance is shown in Figure 9-17

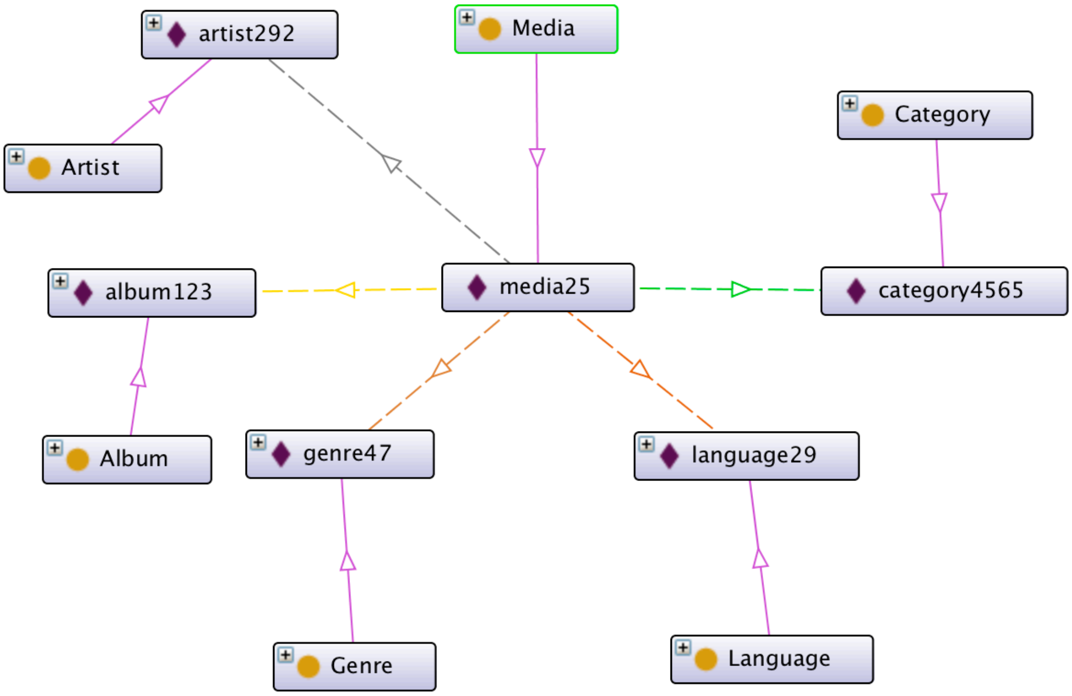


Figure 9-17 Media instance example

7. User Class

This class is to represent the users that the system will recommend items to. The data is taken from a relational database table, MEDIARATING[1]. To transfer the content of these two tables into RDF format, we do the same as we have done with the *Language* table.

8. Rating Class

This class is to represent the ratings of the users to the items. Its data is coming from MEDIARATING class. TIMWE uses 5 stars schema. We created the value of each rating simply by dividing the rating value by 5.

After finishing the Ontology Populating process, we got 3.5 million RDF triples; their size is 4 GB

9.1.3 Configuration process

We created the Joined Ontology by adding two PropertySimilarity instances; the first one is applied on *to:hasArtist* predicate with 0.75 similarity value, and the second one is applied on *to:hasAlbum* with 0.3 similarity value.

[1] We extracted user's info from the rating table because TIMWE does not have any demographic information for the users rather than the mobile number, and we just care about the users who have rated some items.

9.1.4 Testing the proposal

We uploaded the data to Fuseki after increasing the heap size from 1GB to 6 GB. The test was done on a MacBook Pro machine, processor 2,5 GHz Intel Core i7, Memory 16 GB 1600 MHz DDR3.

With 3.5 million triples, Fuseki could not run with the reasoner on. So, we have tried to upload all the triples without any reasoner, which means we had to manipulate the queries that depend on the reasoner to infer some triples.

With 3.5 million triples, Fuseki could not generate justifications because justifications are being created using extensive SPARQL operations such as bind [1], straafter [2], and concat [3]

1. *Level 0 instance level*

Running the Level 0 Instance level service, we get the results illustrated in Figure 9-18

	likedItem	suggestedItem	similarity	becauseOf	finalSimilarity
1	to:Media4309242	to:Media4310872	"0.75"^^xsd:float		"0.5"^^xsd:float
2	to:Media4309242	to:Media4309194	"0.75"^^xsd:float		"0.5"^^xsd:float
3	to:Media4309242	to:Media4309470	"0.75"^^xsd:float		"0.5"^^xsd:float
4	to:Media4309242	to:Media4310602	"0.75"^^xsd:float		"0.5"^^xsd:float
5	to:Media4309242	to:Media4309726	"0.75"^^xsd:float		"0.5"^^xsd:float
6	to:Media4309242	to:Media4310654	"0.75"^^xsd:float		"0.5"^^xsd:float
7	to:Media4309242	to:Media4310670	"0.75"^^xsd:float		"0.5"^^xsd:float
8	to:Media4309242	to:Media4106962	"0.75"^^xsd:float		"0.5"^^xsd:float
9	to:Media4309242	to:Media4310928	"0.75"^^xsd:float		"0.5"^^xsd:float
10	to:Media4309242	to:Media4309210	"0.75"^^xsd:float		"0.5"^^xsd:float
11	to:Media4309242	to:Media4310130	"0.75"^^xsd:float		"0.5"^^xsd:float
12	to:Media4309242	to:Media4311016	"0.75"^^xsd:float		"0.5"^^xsd:float
13	to:Media4309242	to:Media4309538	"0.75"^^xsd:float		"0.5"^^xsd:float
14	to:Media4309242	to:Media4310746	"0.75"^^xsd:float		"0.5"^^xsd:float
15	to:Media4309242	to:Media4310240	"0.75"^^xsd:float		"0.5"^^xsd:float
16	to:Media4309242	to:Media4310512	"0.75"^^xsd:float		"0.5"^^xsd:float
17	to:Media4309242	to:Media4309672	"0.75"^^xsd:float		"0.5"^^xsd:float

Figure 9-18 TIMWE Level 0 Instance experiment

2. *Level 1 Instance level*

We could not run the Level 1 Instance level service with 3.5 million triples, Fuseki would crash though we have used 6GB heap size instead 1GB and we run a TDB service without a reasoner. We run Fuseki with just 1.5 million triples and we could generate recommendation for the mentioned service as illustrated in Figure 9-19

[1] https://www.w3.org/TR/sparql11-query/#bind
[2] https://www.w3.org/TR/sparql11-query/#func-strafter
[3] https://www.w3.org/TR/sparql11-query/#func-concat

	item	finalSimilarity	becauseOf
1	to:Media4310130	"0.25"^^xsd:float	
2	to:Media4310240	"0.25"^^xsd:float	
3	to:Media4310872	"0.25"^^xsd:float	

Figure 9-19 TIMWE Level 1 Instance experiment

3. *User Context service*

In order to run a User Context service, we need to have a UserContext instance, which should be applied on a subclass of the User class. Thus, we need a subclass of the User class. Creating a class with some OWL restrictions can do that. However, TIMWE's data does not have any attributes for the users, and even if it had, we would need the reasoner to be on in order to infer the users that are from the type of the User Context class.

4. *Other services*

To run other services, such as boosting service, we would need to define extra classes, such as classes for Boosting. However, that means we need the reasoner to be on so it can infer that an item belongs to a Boosting class.

In conclusion, the quality of TIMWE data was not enough to make a test and evaluate the Semantic Recommender.

9.2 Movielens Case Study

MovieLens[1] is a recommender system for movies. It provides a dataset for CF systems. The dataset is about movies. Description about it is provided in the next section.

9.2.1 Dataset Description

The dataset contains the following datasets:

1. u.user dataset: it has demographic information about the users; this is a tab-separated list of user id, age, gender, occupation, and zip code.
2. u.genre dataset: it has a list of the genres, with id and name for each genre.

1. u.item dataset: it has information about the items (movies); this is a tab-separated list of movie id, movie title, release date, video release date, IMDb URL, unknown, Action, Adventure, Animation, Children, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, and

[1] https://movielens.org/

Western. The last 19 fields are the genres, a 1 indicates the movie is of that genre, a 0 indicates it is not; movies can be in several genres at once.

2. u.data dataset: it has the full u data set, 100000 ratings by 943 users on 1682 items. Each user has rated at least 20 movies. Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of user id, item id, rating, and timestamp. The time stamps are Unix seconds since 1/1/1970 UTC.

9.2.2 Ontology Creating Process

We developed a small Ontology for that dataset. The important information in the dataset is the ratings and the genre of the movies. The other information is not relevant for item similarities.

Figure 9-20 shows the class diagram of the MovieLens Ontology.

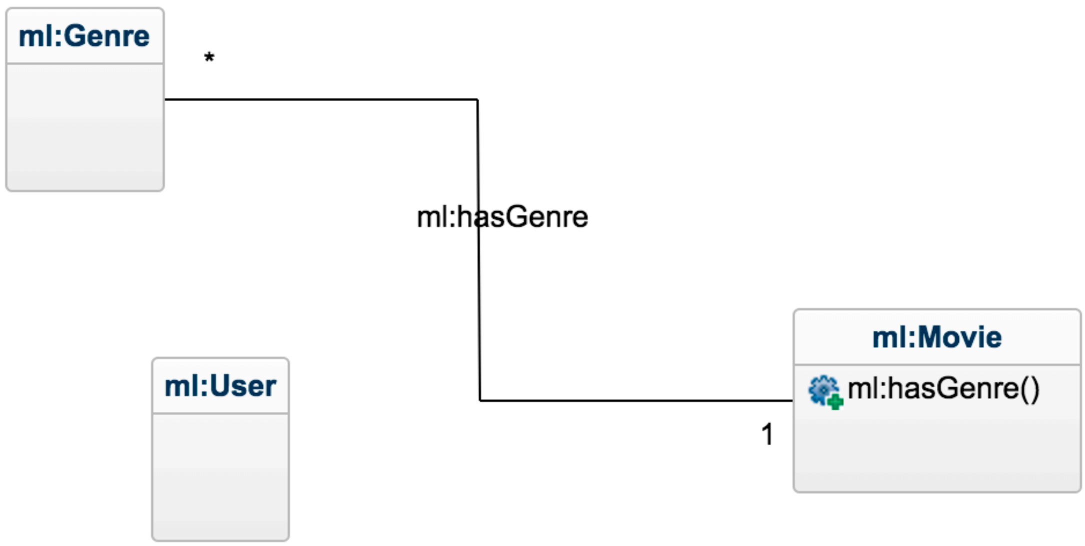


Figure 9-20 Movielens Class Diagram

9.2.3 Ontology Populating Process

In this process, we have built some scripts[1] to extract the data from the datasets and create RDF triples. The result is 370000 triples.

9.2.4 Configuration Process

We created the Joined Ontology with one Property Similarity instance; it is about *hasGenre* predicate with 70% value.

[1] Python scripts

9.2.5 Testing The Propose Solution

As in TIMWE's case, Fuseki couldn't handle the data when the reasoned is on. Thus, we turned the reasoned off. To handle that, we modified the Ontology Populating Process to create the instances off the *rs:Likes* class.

Running the proposal with instance 0 level similarities for the user *ml:user5* gives recommendations as illustrated in Figure 9-21, Figure 9-22, Figure 9-23, Figure 9-24, and Figure 9-25

Recommended Item	Recommended Item Similarity	Reason
movie_184	13.533329	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_horror for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_horror for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_173	11.479998	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_romance for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_romance for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_romance for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_romance for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_110	11.479998	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_war for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_war for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_war for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_war for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174

Figure 9-21 Movielens Level 0 Instance experiment

movie_21	11.386664	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_thriller for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_musical for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_thriller for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_musical for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_50	10.826665	it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_romance for predicate hasGenre with movie_172 ,and it shares genre_war for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_romance for predicate hasGenre with movie_181 ,and it shares genre_war for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_war for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_romance for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_172	10.733332	it shares genre_drama for predicate hasGenre with movie_135 ,and it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_romance for predicate hasGenre with movie_50 ,and it shares genre_war for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_romance for predicate hasGenre with movie_181 ,and it shares genre_war for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_war for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_romance for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174

Figure 9-22 Movielens Level 0 Instance experiment continue 1

movie_181	10.359999	it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_romance for predicate hasGenre with movie_172 ,and it shares genre_war for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_romance for predicate hasGenre with movie_50 ,and it shares genre_war for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_war for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_romance for predicate hasGenre with movie_70 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_164	10.079999	it shares genre_thriller for predicate hasGenre with movie_135 ,and it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_thriller for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_101	9.613333	it shares genre_animation for predicate hasGenre with movie_169 ,and it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_animation for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_horror for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_animation for predicate hasGenre with movie_189 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174

Figure 9-23 Movielens Level 0 Instance experiment continue 2

movie_17	9.5199995	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_thriller for predicate hasGenre with movie_135 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_horror for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_crime for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_thriller for predicate hasGenre with movie_183 ,and it shares genre_horror for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_crime for predicate hasGenre with movie_194 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_action for predicate hasGenre with movie_174
movie_62	9.333333	it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_82	9.333333	it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_adventure for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_adventure for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_adventure for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_adventure for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_adventure for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_adventure for predicate hasGenre with movie_174 ,and it shares genre_action for predicate hasGenre with movie_174
movie_176	9.146667	it shares genre_thriller for predicate hasGenre with movie_135 ,and it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_war for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_war for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_thriller for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_war for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_war for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_action for predicate hasGenre with movie_174

Figure 9-24 Movielens Level 0 Instance experiment continue 3

movie_74	8.026668	it shares genre_comedy for predicate hasGenre with movie_169 ,and it shares genre_drama for predicate hasGenre with movie_135 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_drama for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_comedy for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_comedy for predicate hasGenre with movie_95 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_comedy for predicate hasGenre with movie_186 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_comedy for predicate hasGenre with movie_194 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_comedy for predicate hasGenre with movie_189 ,and it shares genre_comedy for predicate hasGenre with movie_163 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_comedy for predicate hasGenre with movie_70 ,and it shares genre_action for predicate hasGenre with movie_174
movie_39	7.4666677	it shares genre_sciFi for predicate hasGenre with movie_135 ,and it shares genre_action for predicate hasGenre with movie_101 ,and it shares genre_sciFi for predicate hasGenre with movie_101 ,and it shares genre_action for predicate hasGenre with movie_172 ,and it shares genre_sciFi for predicate hasGenre with movie_172 ,and it shares genre_action for predicate hasGenre with movie_50 ,and it shares genre_sciFi for predicate hasGenre with movie_50 ,and it shares genre_action for predicate hasGenre with movie_29 ,and it shares genre_crime for predicate hasGenre with movie_29 ,and it shares genre_action for predicate hasGenre with movie_183 ,and it shares genre_sciFi for predicate hasGenre with movie_183 ,and it shares genre_action for predicate hasGenre with movie_186 ,and it shares genre_crime for predicate hasGenre with movie_194 ,and it shares genre_action for predicate hasGenre with movie_181 ,and it shares genre_sciFi for predicate hasGenre with movie_181 ,and it shares genre_action for predicate hasGenre with movie_121 ,and it shares genre_sciFi for predicate hasGenre with movie_121 ,and it shares genre_action for predicate hasGenre with movie_174

Figure 9-25 Movielens Level 0 Instance experiment continue 4

To evaluate the system, we build a *Test Simulator* that requests the recommendations for all the users. We choose Mean Reciprocal rank measure, which was described previously at section 5.4.1, because at the end of any recommender system, we care the most about if the user likes the recommendations. To use that measure, we divided the dataset u.data, which was described previously in section 9.2.1, to two datasets; one for training (80% of the original dataset), and one for testing (20% of the original dataset). The training set contains 80000 ratings, while the testing dataset contains 20000 ratings.

The Test Simulator would run the Level0Instance service for each user, and then compares the recommendations list returned with the testing set. The Test Simulator finds the first item in the recommendations list that the user has already liked. (we know that the user has liked it using the testing set).

For instance, the users that their first recommended item was good[1] are[2]:

5 ,10 ,17 ,23 ,38 ,41 ,43 ,45 ,56 ,64 ,83 ,95 ,96 ,106 ,109 ,117 ,125 ,128 ,148 ,157 ,158 ,160 ,189 ,194 ,222 ,223 ,232 ,246 ,247 ,250 ,274 ,280 ,287 ,340 ,348 ,350 ,374 ,430

The users that their second recommended item was good are:

26 ,28 ,53 ,74 ,104 ,108 ,113 ,131 ,177 ,186 ,218 ,239 ,248 ,252 ,292 ,323 ,328 ,329 ,367 ,390 ,414

The users the their third recommend item was good are:

1 ,6 ,11 ,14 ,21 ,24 ,49 ,57 ,58 ,62 ,69 ,72 ,73 ,75 ,76 ,79 ,99 ,114 ,119 ,136 ,138 ,168 ,176 ,193 ,195 ,213 ,221 ,235 ,237 ,244 ,249 ,251 ,257 ,269 ,306 ,313 ,315 ,320 ,321 ,325 ,331 ,338 ,342 ,345 ,361 ,381 ,401 ,447

<u>The recommender quality according to Mean Reciprocal rank is 47%.</u>

Though we knew just one feature about the movies, which is the genre of it, we could get recommendations with good accuracy results. If there were more features, the system would have got better results.

To improve the results, we tried to get more information for the movies using IMDB API. We have done that by writing Python scripts, during the *Configuration* process that iterates over the existing movies fetching their information from IMDB and create RDF triples. The info that we got for each movie is: the first three staring actors of the movie [3], the directors, and the writers. We modified the Ontology, illustrated in Figure 9-20, by creating new classes and new predicates as illustrated in Figure 9-26.

In the *Configuration* process, we created three *Important Property* instances; one for *ml:writtenBy* predicate with 55% similarity value, one for *ml:hasStar* predicate with 60% similarity value, and one for *ml:directedBy* predicate with 50% similarity value.

The recommender quality according to Mean Reciprocal rank becomes 55%

Though the ratings provided by MovieLens are random, the quality increases when the data is modeled better. Moreover, the number of people whose recommendations were

[1] Good means that they have rated it 4 or 5 starts in the test dataset.

[2] There are just the users' IDs from the training dataset

[3] Getting the first three stars of the movie does not necessary mean getting them in order, that is because the IMDB API crawls the IMDB website, and the IMDB does not necessary have the actors ordered according to their priority for the movie.

already in the test set increased as well. In other words, to apply the Mean Reciprocal rank metric, we need to know in advance if the user likes the item or not, and that is why we divided the data for test and prepare datasets. However, some users got recommendations that the test data does not have information whether they liked those items or not. Without IMDB, the number of users were 210, which is equal to 20% of the users, while with IMDB, the number of users became 298, which is equal to 30% of the users.

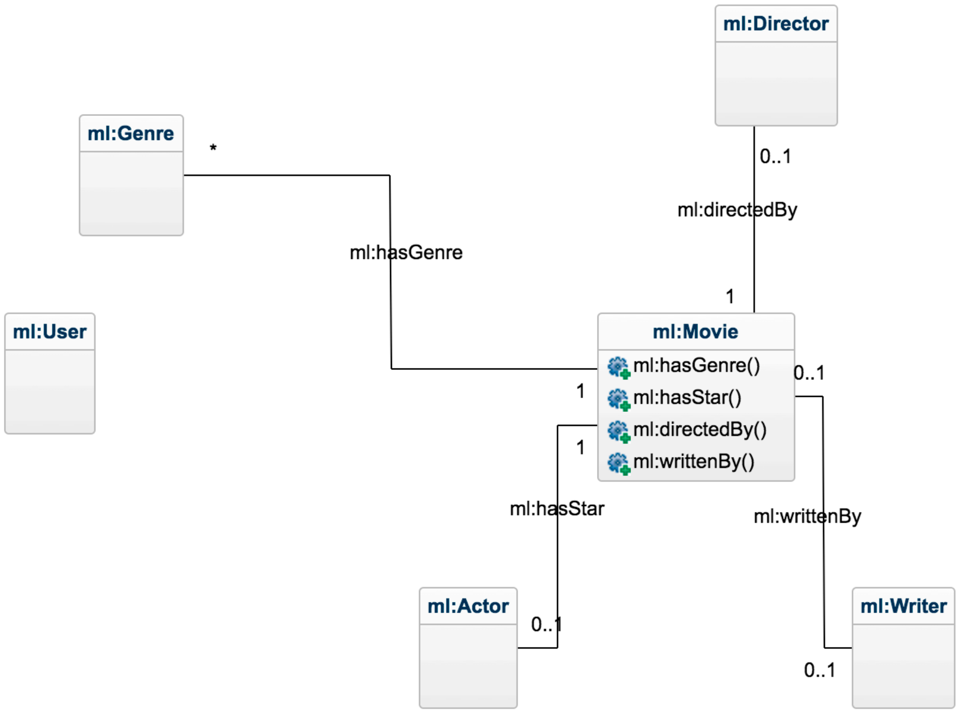


Figure 9-26 Movielens Class Diagram With IMDB

9.3 Other Domains

9.3.1 Book Domain

A research [60] provided a dataset from *bookcrossing* website[1], which has the following datasets:

- BX-Users dataset: it has demographic information about the users; this is a tab-separated list of user id, location, and age.
- BX-Books dataset: it has information about the items (books); this is a tab-separated list of ISBN, Book-title, Book-author, year of publication, and publisher.

[1] http://www.bookcrossing.com/

- BX-Book-Ratings dataset: it has users' ratings; this is a tab-separated list of User-ID, ISBN, and Book-rating value. They have used a 10 starts rating schema.

We do not have any important attributes for the book. We tried to use DBpedia to get some information about the books. We searched for the books that the dataset has in DBpedia using ISBN or the book's title, but DBpedia does not have any of them.

We tried to use Google Books API to get some information about the books, but the available information are not descriptive data. For example, checking the available content for *Mountain View* book which its ISBN is 0738531367, using this API https://www.googleapis.com/books/v1/volumes?q=isbn:0738531367 gives attributes for the publisher, the published date, number of pages, category, and image links, which are not important from similarity point of view[1].

9.3.2 Wine Ontology

It provides former description of the wine, its fruits, its vintage's years, its regions, and the wineries [2]. Figure 9-27 shows the classes of the Wine Ontology, while Figure 9-28 shows the OWL object properties for it.

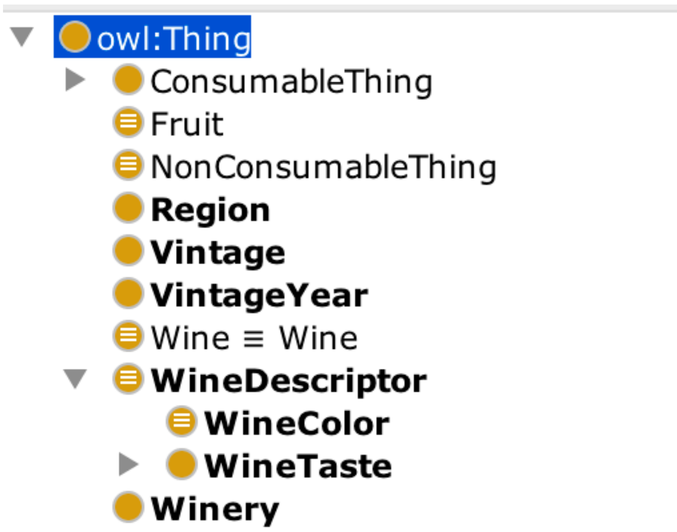


Figure 9-27 Wine Ontology Classes

[1] Some would argue that category is important, but who likes History books can also like Music books. What we were looking for is a good-enough descriptive attributes.
[2] It can be downloaded from this URL: https://www.w3.org/TR/owl-guide/wine.rdf

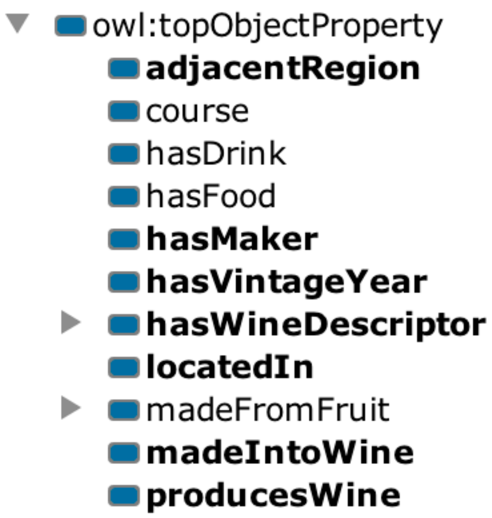


Figure 9-28 Wine Ontology Object Properties

Wine Ontology is supposed to be a sample example of OWL full reasoning, that's why Fuseki couldn't run it with the reasoner.

9.3.3 DBpedia Music Ontology

Looking at the DBpedia data, we could find some information about composers such as: Baroque Composers[1], Classical Era Composers[2], English Composers[3], German Composers[4], Italian Composers[5], Opera Composers[6], Romantic Composers[7], Women Classical Composers[8], and 18th Century Classical Composers[9]. It also contains some information about musical players such as: German Clarinetists[10], German Classical Oboists[11], German Drummers[12], German Flautists[13], German Guitarists[14], and German

[1] http://dbpedia.org/class/yago/BaroqueComposers
[2] http://dbpedia.org/class/yago/ClassicalEraComposers
[3] http://dbpedia.org/class/yago/EnglishComposers
[4] http://dbpedia.org/class/yago/GermanComposers
[5] http://dbpedia.org/class/yago/ItalianComposers
[6] http://dbpedia.org/class/yago/OperaComposers
[7] http://dbpedia.org/class/yago/RomanticComposers
[8] http://dbpedia.org/class/yago/WomenClassicalComposers
[9] http://dbpedia.org/page/Category:18th-century_classical_composers
[10] https://en.wikipedia.org/wiki/Category:German_clarinetists
[11] http://dbpedia.org/page/Category:German_classical_oboists
[12] https://en.wikipedia.org/wiki/Category:German_drummers
[13] https://en.wikipedia.org/wiki/Category:German_flautists
[14] https://en.wikipedia.org/wiki/Category:German_guitarists

Pianists[1]. Plus, we could get some symphonies' information such as Choral Symphony[2], Romantic Symphony[3].

We used a manual approach in the Ontology Creating Process, shown in Figure 7-1, and the resulting Ontology classes are illustrated in

Figure 9-29 DBpedia Music Ontology Classes

1 http://live.dbpedia.org/page/Category:German_pianists
2 http://dbpedia.org/data/Category:Choral_symphonies.ntriples
3 https://en.wikipedia.org/wiki/Category:Romantic_symphonies

In the *Ontology Populating Process* we added ABox triples by making SPARQL queries to DBpedia endpoint and extract the resulting RDF triples and then insert them to the Domain Triple Store.

DBpedia does not having information about symphonies such as features, keys, or harmonies. It doesn't have important features for the composers. Moreover, there are no ratings. That's why we couldn't use DBpedia Ontology.

9.3.4 Jokes Domain

Juster[1] provides a dataset. However, it is intended from CF approach since it just has the user id, joke id and rating value.

9.4 Conclusion

In this chapter, an Ontology for TIMWE was created, and the Semantic Recommender was tested with it. We got results, but we were not able to evaluate the recommender.

A movie's Ontology was created and ABox axioms were generated, and the Semantic Recommender tested on it, we got results. The quality of the recommender according to the *Mean Reciprocal Rank* metric is 47%. After improving the data, the quality becomes 55%. The *Support* metric shows improving from 20% to 30%.

[1] http://goldberg.berkeley.edu/jester-data/

References

[1] M. Kleerebezem and L. Quadri, Peptide pheromone-dependent regulation of antimicrobial peptide production in Gram-positive bacteria: a case of multicellular behavior. 2001.
[2] NC State University, The Honey Bee Dance Language. .
[3] M. Bilandzic and M. Foth, “Social Navigation and Local Folksonomies: Technical and Design Considerations for a Mobile Information System.” .
[4] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, “WTF: The Who to Follow Service at Twitter.” .
[5] M. DESHPANDE and G. KARYPIS, “Item-Based Top-N Recommendation Algorithms.” University of Minnesota.
[6] S.-T. Park, D. Pennock, O. Madani, N. Good, and D. DeCoste, “Naıve Filterbots for Robust Cold-Start Recommendations.” .
[7] F. McSherry and I. Mironov, “Differentially Private Recommender Systems:” .
[8] J. Beel, S. Langer, A. Nürnberger, and M. Genzmehr, “The Impact of Demographics (Age and Gender) and Other User-Characteristics on Evaluating Recommender Systems.” .
[9] K. Sakamoto, “Cultural Influence to the Color Preference According to Product Category.” .
[10] G. Ngai, S. Chi-fai Chan, and Y. Wang, “Applicability of Demographic Recommender System to Tourist Attractions: A Case Study on Trip Advisor.” .
[11] I. Sparling and S. Sen, “Rating: How Difficult is It?” .
[12] M. Pazzani and D. Billsus, “Content-based Recommendation Systems.” .
[13] P. Lops, M. Gemmis, and G. Semeraro, “Content-based Recommender Systems: State of the Art and Trends.” .
[14] G. Salton, Wong, and C. S. Yang, “A vector space model for automatic indexing.” .
[15] T. Kamba, K. Bharat, and M. Albers, “The Krakatoa Chronicle - An Interactive, Personalized, Newspaper on the Web http://www.w3.org/Conferences/WWW4/Papers/93.”
[16] M. Howe, “Pandora’s Music Recommender.” .
[17] M. Balabanovic and Y. Shoham, Fab, Content-based, collaborative recommendation. .
[18] McCallum and Nigam, “A Comparison of Event Models for Naive Bayes Text Classification.” .
[19] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, “Using collaborative filtering to weave an information Tapestry.” .
[20] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Item-Based Collaborative Filtering Recommendation Algorithms.” GroupLens Research Group/Army HPC Research Center Department of Computer Science and Engineering University of Minnesota, Minneapolis, MN 55455.
[21] X. Su and T. M. Khoshgoftaar, “A Survey of Collaborative Filtering Techniques.” 03-Aug-2009.
[22] A. M. Rashid, G. Karypis, and J. Riedl, “Learning Preferences of New Users in Recommender Systems: As Information Theoretic Approach.” .
[23] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems.” 2002.
[24] M. Berry, S. Dumais, and G. O’Brien, “Using linear algebra for intelligent information retrieval.” Dec-1995.
[25] Claypool, Gokhale, and Miranda, “Combining content-based and collaborative filters in an online newspa- per.” .
[26] S. Lam and J. Riedl, “Shilling Recommender Systems for Fun and Profit.” .

[27] P. Melville and V. Sindhwani, "Recommender Systems." IBM T. J. Watson Research Center,.
[28] J. Herlocket, J. Konstan, Al Borchers, and J. Riedl, "An Algorithmic Framework for Performing Collaborative Filtering." .
[29] M. Ekstrand, J. Riedl, and J. Konstan, "Collaborative Filtering Recommender Systems." .
[30] B. Kitts, D. Freed, and M. Vrieze, "Cross-sell: A Fast Promotion-Tunable Customer-item Recommendation Method Based on Conditionally Independent Probabilities." .
[31] S. Vucetic and Z. Obradovic, "Collaborative Filtering Using a Regression-Based Approach." .
[32] S. McNee, J. Riedl, and J. Konstan, "Being Accurate is Not Enough: How Accuracy Metrics have hurt Recommender Systems." .
[33] A. Gunawardana and G. Shani, "A Survey of Accuracy Evaluation Metrics of Recommendation Tasks." .
[34] M. Christopher, P. Raghavan, and H. Schütze, Evaluation in information retrieval (Chapter 8). 2009.
[35] T. Nathanson, Algorithms, Models and Systems for EigentasteBased Collaborative Filtering and Visualization. Electrical Engineering and Computer Sciences University of California at Berkeley, 2009.
[36] T. Nathanson, E. Bitton, and K. Goldberg, "Eigentaste 5.0: Constant-Time Adaptability in a Recommender System Using Item Clustering." .
[37] Y. Shi, A. Karatzoglou, and L. Baltrunas, "CLiMF: Learning to Maximize Reciprocal Rank with Collaborative Less-is-More Filtering." .
[38] A. Ginige and J. Robertson, "Hypermedia Authoring." .
[39] Lowe and Hall, "Hypermedia and the Web. Conference ,1999."
[40] C. Goble, L. Carr, D. De Roure, and W. Hall, "Conceptual Open Hypermedia = The Semantic Web?" .
[41] T. Berners-Lee, H. James, and L. Ora, "The Semantic Web, Scientific American Magazine, May 17, 2001."
[42] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax." Aug-1998.
[43] T. Berners-Lee, "Linked Data http://www.w3.org/DesignIssues/LinkedData.html." 27-Jul-2006.
[44] N. Guarino, D. Oberle, and S. Staab, What Is an Ontology? .
[45] Dan Brickley, Guha, and Brian McBride, "RDF Schema 1.1 http://www.w3.org/TR/rdf-schema/."
[46] I. Herman, B. Adida, and M. Sporny, "RDFa 1.1 Primer - Third Edition." .
[47] F. Gandon and G. Schreiber, "RDF 1.1 XML Syntax." .
[48] D. Beckett, T. Berners-Lee, and G. Carothers, "RDF 1.1 Turtle." .
[49] D. Beckett, "RDF 1.1 N-Triples." .
[50] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD 1.0." .
[51] W3C OWL Working Group, "OWL 2 Web Ontology Language https://www.w3.org/TR/owl2-overview/." .
[52] Bahramiana and Abbaspoura, An Ontology-based tourism recommender system based on spreading activation model. .
[53] Middleton and Shadbolt, Ontological User Profiling in Recommender Systems. .
[54] M. Gan, X. Dou, and R. Jiang, "From Ontology to Semantic Similarity: Calculation of Ontology-Based Semantic Similarity." .

[55] Rada, Mili, Bicknell, and Blettner, “Development and application of a metric on semantic nets.”
[56] Sussna, Word sense disambiguation for free-text indexing using a massive semantic network. .
[57] Tversky, Features of similarity. .
[58] V. CODINA and L. CECCARONI, Taking Advantage of Semantics in Recommendation Systems. .
[59] M. Arenas, A. Bertails, E. Prud’hommeaux, and J. Sequeda, “A Direct Mapping of Relational Data to RDF.” 27-Sep-2012.
[60] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, Improving Recommendation Lists Through Topic Diversification. .
[61] Anderson, Bai, Bischof, Blackford, Demmel, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney, and Sorensen, LAPACK Users’ Guide., Third Edition. 1999.

10 Annex

10.1 Value Analysis

10.1.1 Canvas Model

<table>
<tr>
<td rowspan="2">Key Partners
The SPARQL server provider: because the Semantic Recommender needs it. It could be just Fuseki or it could be other paid SPARQL server (depending on the choice of the customer).</td>
<td>Key Activities
Assist the customers in building the Joined Ontology.
Assist the customers in modeling their data.</td>
<td>Value Propositions
Generate good recommendations.</td>
</tr>
<tr>
<td>Key Resources
Human Resources: the software team that builds the system, and the support team that will help the customer custom and build the Joined Ontology.
Financial Resources: Any resource that will help us to advertise our product.
Physical Resources: The servers that we use to present demos to the customer.</td>
<td>Customer Segments
The customers are any company that wants to create recommendations about its products to its clients. The company has to provide its clients a way to rate its products.</td>
</tr>
<tr>
<td>Cost Structure
Salaries of the knowledge engineers.
The cost of the demo servers</td>
<td>Revenue Streams
Selling the Semantic Recommender</td>
<td>Customer Relationships
Dedicated assistance by providing a knowledge engineer who helps configuring the Semantic Recommender.</td>
</tr>
</table>

10.1.2 SWOT Analysis

Strengths	**Weakness**
It works with any domain It provides better recommendations when the	Some manual work to configure the Joined Ontology.

domain data is modeled enough.	Lake of knowledge, and thus trust, in The Semantic Web technologies.
Opportunities	**Threats**
The ability to take advantage of Linked Data, which enriches the domain data.	Most of the domain systems don't have a corresponding Ontology. Emerging problems in the SPARQL endpoint when handing large amount of data because they are still not mature enough comparing to traditional Relational Database servers.

10.2 SVD Example

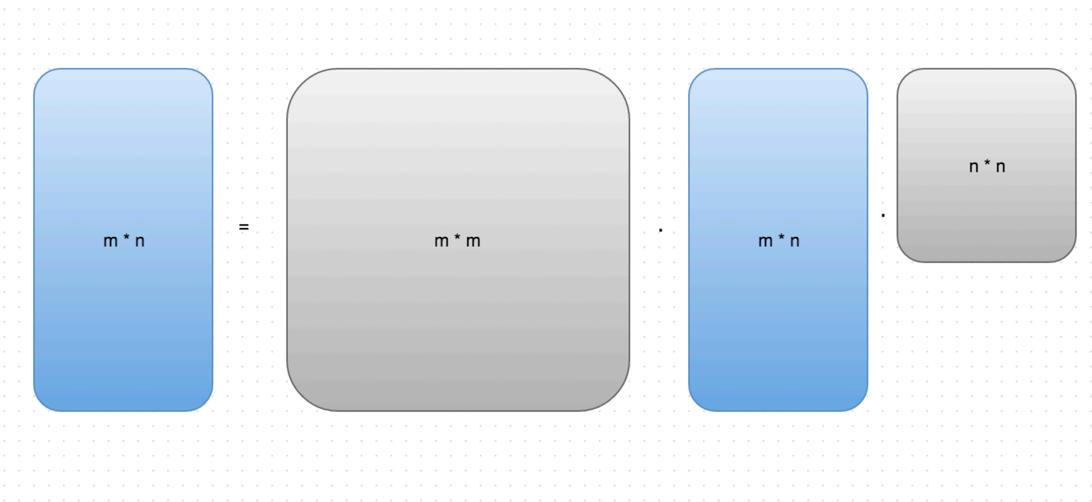


Figure 10-1 SVD equation

SVD chops these matrices like this:

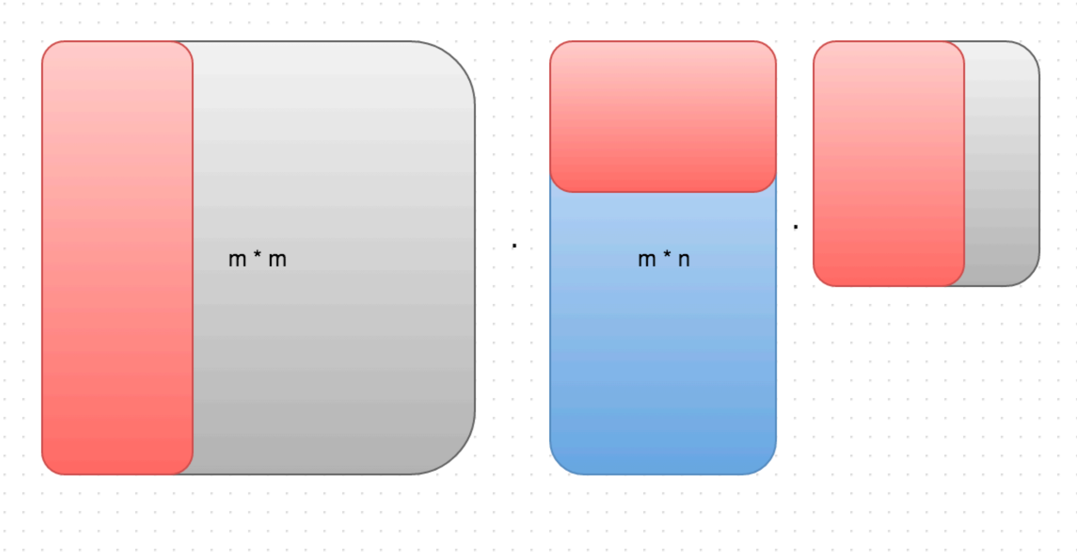


Figure 10-2 SVD Chops

And we get to decide how far we want to chop the matrices (that is called K), then we work with just the chops areas, which are way less smaller than the original matrix. As a result, instead of $m \times n$ matrix, we will have $m \times k$ matrix

Let us say that we have the following ratings:

Table 10-1 Sample data for SVD

	Francesco Totti	Sarah	Maria	Edison	William
Product1	5	2	1	1	4
Product2	0	1	1	0	3
Product3	3	4	1	0	2
Product4	4	0	0	0	3
Product5	3	0	2	5	4
Product6	2	5	1	5	1

Looking at the ratings, we can see that William is user most close to Francesco Totti. Let us apply SVD to see if that is still the case:

Using R algorithm to calculate SVD [61] we get

Table 10-2 The value of S matrix

12.6517062	5.7708678	4.7486887	2.5785141	0.6577509

Table 10-3 The value of U matrix

-0.5096819	0.37471910	0.163866425	-0.14236469	-0.07373153
-0.1730378	0.13786697	0.009794465	0.91117680	0.34693168
-0.3629127	0.01699667	0.619884038	0.13430353	-0.53516354
-0.3093885	0.51443751	0.011349117	-0.33259076	0.54067263
-0.5064356	-0.03881341	-0.753794194	0.06155236	-0.36746157
-0.4757924	-0.75771827	0.143042486	-0.13056184	0.40055855

Table 10-4 The value of V^T matrix

-0.5806010	0.40729728	0.1577453	-0.6653997	-0.17137999
-0.3970227	-0.49096593	0.7438424	0.1981189	0.09366344
-0.2003126	-0.05298377	-0.1202439	0.3473545	-0.90661669
-0.4284657	-0.62519874	-0.6085663	-0.1890283	0.13949559
-0.5306293	0.44652059	-0.1923625	0.6013363	0.34704930

Let us say we choose K equals to 2, which means we will take the first two columns and drop the others off. In other words, we think that the large amount of the information has been captured in the first two columns.

Table 10-5 The value of S after chopping K = 2

12.6517062	0
0	5.7708678

Table 10-6 The value of U matrix after chopping K = 2

-0.5096819	0.37471910
-0.1730378	0.13786697
-0.3629127	0.01699667
-0.3093885	0.51443751
-0.5064356	-0.03881341
-0.4757924	-0.75771827

Table 10-7 The value of V^T matrix after chopping K = 2

-0.5806010	0.40729728	Francesco Totti
-0.3970227	-0.49096593	Sarah

-0.2003126	-0.05298377	Maria
-0.4284657	-0.62519874	Edison
-0.5306293	0.44652059	William

Obvious that Francesco, who is represented by the first line of V^T **,** is the most user close to Francesco Totti, who is represented by the fifth line of V^T**.**

Note that in a real live example, we could use the vector space to calculate the angle between the users in order to find the closest users.

10.3 Term Frequency – Inverse Document Frequency (TF-IDF)

It is a measure to calculate the weight of a term in a document. It is used to create users' profiles. Knowing that a user likes a document is not sufficient to build his/her profile, we need to extract features (terms) for that documents and assign these features a weight corresponding to their representation of these documents. Then, we can add these terms to users' profiles. The purpose of TF-IDF is to estimate the weight of these terms related to the document, taking into consideration the other documents, which are called items in the context of RS, the information repository has.

The use of TF-IDF appeared mainly in Search Engine context, where users input a list of terms as a query, and the system responses by the documents that contain these terms. However, the problem starts to appear when there are too many documents, which the system must rank and evaluates how close they are to the query.

To reach that purpose, we need to take into account some considerations. Such as:

1. How many times the term has appeared in the document.
2. How many times the term has appeared in other documents.

The meaning of TF-IDF is as the following:

1. Term Frequency (TF): the number of times a term has appeared in a document. Even if the items are not unstructured text, we can still use TF. For instance, in a movie context, TF could be the number of times the community has assigned a tag, such as "*the importance of education"*, to *The Reader* movie.
2. Document Frequency (DF): the number of documents in the corpus in which the term appeared at least once. In a movie context, DF could be the number of movies that were assigned to the tag "*the importance of education*".

3. Inverse Document Frequency (IDF): it represents how rare it is for a document to have this term. In a movie context, how rare it is for a movie to have this tag. It can be calculated as: $\frac{n}{DF}$ where n is the number of documents.

The intuition of using TF-IDF is: if a term appears in almost all the documents, so it is not important. However, if it appears in just one (or relatively small number) document, it is important. In other words, the higher the number of documents contain that term, the lower the importance of that term to differentiate these documents. So, we care a lot about rare terms.

Mathematicians have suggested many modifications and variants to calculate TF, DF, and IDF, as illustrated in the next section.

10.3.1 Variants

Scientists populated TF-IDF to optimize the performance for a specific domain. Some of them are:

1. Some researchers suggest Boolean to represent the TF.
2. Some researchers suggest doing the logarithmic to the TF because the TF seems to be a big number. Log (tf+1). They added one to avoid log (0)
3. Normalize the TF because we could end up with the following example. A book has a term 8 times while a small paragraph has that term 4 times.
4. Another alternative to TF-IDF is BM25

In the next section, we discuss how to apply TF-IDF in the context of CB.

10.3.2 TF-IDF in CB

The purpose of TF-IDF in the CB approach is to build users' profiles. So, it's being applied to the items (such as documents, or movies), in order to calculate the weight of their features (such as terms or tags), and when a user states (explicitly or implicitly) that he likes, or dislikes, that item, RS will add the weights of the items' features to the user's profile.

10.3.3 Drawbacks

1. It does not handle phrases and n gram, which is a sequence of n terms that form an expression. For instance, the phrase *computer science* is different than *computer* and *science*.
2. It does not consider the context of the terms.
3. It does not give the title and headlines more priorities than the other text.

10.4 SPARQL Queries

10.4.1 Level 0 and 1 Instance and Class service

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX <RECOMENDERPREFIX>: <<RECOMMENDERURI>>
PREFIX <DOMAINPREFIX>: <<DOMAINRUI>>
PREFIX  <JOINEDONTOLOGYRPREFIX>: <<JOINEDONTOLOGYURI>>


SELECT ?recommendedItem (SUM(?finalSimilarity) AS
?recommendedItemSimilarity) (GROUP_CONCAT (?becauseOf ; separator=' ,and ') AS
?reason)
WHERE
  { SELECT DISTINCT  (?item AS ?recommendedItem) (( ( ?similarity * ?importance
) * ?levelImportance * ?ratingValue) AS ?finalSimilarity) ?becauseOf
    WHERE
      { VALUES ?user { <USERURI> }
       ?user     rs:hasRated          ?ratings .
       ?ratings  rdf:type             rs:Likes ;
                 rs:aboutItem         ?x ;
                 rs:ratesBy           ?ratingValue .
       ?x        rdf:type             ?class .
       ?class    rdfs:subClassOf*     ?mainClass .
       ?mainClass  rdfs:subClassOf*   rs:RecommendableClass ;
                 rs:hasSimilarityConfiguration  ?similarityConfiguration
        { VALUES ?classImportance { <classSimilarityWeight> }
          BIND(?classImportance AS ?importance)
          BIND(( 2 / 3 ) AS ?levelImportance)
          ?similarityConfiguration
                    rs:hasClassSimilarity  ?classSimilarity .
          ?classSimilarity
```

```
            rs:appliedOnClass     ?class ;
            rs:hasClassSimilarityValue  ?similarity .
        ?item     rdf:type             ?class
        BIND(concat("it shares the same class, which is ", strafter(str(?class), "#"), ",
with ", strafter(str(?x), "#")) AS ?becauseOf)
       }
      UNION
       { VALUES ?instanceImportance { <instanceSimilarityWeight> }
        BIND(?instanceImportance AS ?importance)
        BIND(( 2 / 3 ) AS ?levelImportance)
        ?similarityConfiguration
            rs:hasPropertySimilarity  ?propertySimilarity .
        ?propertySimilarity
            rs:appliedOnProperty  ?property ;
            rs:hasPropertySimilarityValue  ?similarity .
        ?x       ?property            ?value .
        ?item     ?property            ?value
        BIND(concat("it shares ", strafter(str(?value), "#"), " for predicate ",
strafter(str(?property), "#"), " with ", strafter(str(?x), "#")) AS ?becauseOf)
       }
      UNION
       { VALUES ?instanceImportance { <instanceSimilarityWeight> }
        BIND(?instanceImportance AS ?importance)
        BIND(( 1 / 3 ) AS ?levelImportance)
        ?similarityConfiguration
            rs:hasPropertySimilarity  ?propertySimilarity ;
            rs:hasPropertySimilarity  ?propertySimilarity2 .
        ?propertySimilarity
            rs:appliedOnProperty  ?property .
        ?propertySimilarity2
            rs:appliedOnProperty  ?property2 .
        ?propertySimilarity
            rs:hasPropertySimilarityValue  ?similarity0 .
        ?propertySimilarity2
```

```
                    rs:hasPropertySimilarityValue  ?similarity .
          ?x      ?property         ?y .
          ?item    ?property         ?z .
          ?y      ?property2        ?f .
          ?z      ?property2        ?f
          FILTER ( ?y != ?z )
          BIND(concat("Both of ", strafter(str(?y), "#"), " and ", strafter(str(?z), "#"), " share ", strafter(str(?f), "#"), " for ", strafter(str(?property2), "#")) AS ?becauseOf)
        }
       UNION
        { VALUES ?classImportance { <classSimilarityWeight> }
          BIND(?classImportance AS ?importance)
          BIND(( 1 / 3 ) AS ?levelImportance)
          ?similarityConfiguration
                    rs:hasPropertySimilarity  ?propertySimilarity .
          ?propertySimilarity
                    rs:appliedOnProperty  ?property .
          ?similarityConfiguration
                    rs:hasClassSimilarity  ?classSimilarity .
          ?classSimilarity
                    rs:appliedOnClass     ?targetClass ;
                    rs:hasClassSimilarityValue  ?similarity .
          ?x      ?property         ?y .
          ?item    ?property         ?z .
          ?y      rdf:type          ?targetClass .
          ?z      rdf:type          ?targetClass
          FILTER ( ?y != ?z )
          BIND(concat("both of ", strafter(str(?y), "#"), " and ", strafter(str(?z), "#"), " are from type ", strafter(str(?targetClass), "#")) AS ?becauseOf)
        }
       FILTER ( ?x != ?item )
      }
    }
  GROUP BY ?recommendedItem
```

```
ORDER BY DESC(?recommendedItemSimilarity)
limit <numOfMaxRecommendedItems>
```

10.4.2 User Context and Temporal Context service

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX <RECOMENDERPREFIX>: <<RECOMMENDERURI>>
PREFIX <DOMAINPREFIX>: <<DOMAINRUI>>
PREFIX  <JOINEDONTOLOGYRPREFIX>: <<JOINEDONTOLOGYURI>>



SELECT (?item as ?recommendedItem) (SUM((?userContextWeight * ?temporalContextWeight * ?finalSimilarity )) AS ?recommendedItemSimilarity) (GROUP_CONCAT (DISTINCT ?becauseOf ; separator=' ,and ') AS ?reason) WHERE {
select distinct ?item ?userContextWeight ?userContext ?temporalContextWeight ?itemClassTemporal ( ( ?similarity * ?importance * ?levelImportance * ?ratingValue ) AS ?finalSimilarity) ?becauseOf ?skipUserContext ?skipTemporalContext
{
  {
  VALUES ?user { <USERURI> }
    ?user     rs:hasRated          ?ratings .
    ?ratings  rdf:type             rs:Likes ;
              rs:aboutItem         ?x ;
              rs:ratesBy           ?ratingValue .
    ?x        rdf:type             ?class .
    ?class    rdfs:subClassOf*     ?mainClass .
    ?mainClass  rdfs:subClassOf*   rs:RecommendableClass ;
              rs:hasSimilarityConfiguration  ?similarityConfiguration
     { VALUES ?classImportance { <classSimilarityWeight> }
      BIND(?classImportance AS ?importance)
      BIND(( 2 / 3 ) AS ?levelImportance)
```

```
            ?similarityConfiguration
                    rs:hasClassSimilarity  ?classSimilarity .
            ?classSimilarity
                    rs:appliedOnClass     ?class ;
                    rs:hasClassSimilarityValue  ?similarity .
            ?item     rdf:type             ?class
            BIND(concat("it shares the same class, which is ", strafter(str(?class), "#"), ",
with ", strafter(str(?x), "#")) AS ?becauseOf)
          }
        UNION
          { VALUES ?instanceImportance { <instanceSimilarityWeight> }
            BIND(?instanceImportance AS ?importance)
            BIND(( 2 / 3 ) AS ?levelImportance)
            ?similarityConfiguration
                    rs:hasPropertySimilarity  ?propertySimilarity .
            ?propertySimilarity
                    rs:appliedOnProperty  ?property ;
                    rs:hasPropertySimilarityValue  ?similarity .
            ?x        ?property            ?value .
            ?item     ?property            ?value
            BIND(concat("it shares ", strafter(str(?value), "#"), " for predicate ",
strafter(str(?property), "#"), " with ", strafter(str(?x), "#")) AS ?becauseOf)
          }
        UNION
          { VALUES ?instanceImportance { <instanceSimilarityWeight> }
            BIND(?instanceImportance AS ?importance)
            BIND(( 1 / 3 ) AS ?levelImportance)
            ?similarityConfiguration
                    rs:hasPropertySimilarity  ?propertySimilarity ;
                    rs:hasPropertySimilarity  ?propertySimilarity2 .
            ?propertySimilarity
                    rs:appliedOnProperty  ?property .
            ?propertySimilarity2
                    rs:appliedOnProperty  ?property2 .
```

```
            ?propertySimilarity
                    rs:hasPropertySimilarityValue  ?similarity0 .
            ?propertySimilarity2
                    rs:hasPropertySimilarityValue  ?similarity .
            ?x      ?property          ?y .
            ?item   ?property          ?z .
            ?y      ?property2         ?f .
            ?z      ?property2         ?f
            FILTER ( ?y != ?z )
            BIND(concat("Both of ", strafter(str(?y), "#"), " and ", strafter(str(?z), "#"), "
share ", strafter(str(?f), "#"), " for ", strafter(str(?property2), "#")) AS ?becauseOf)
          }
        UNION
          { VALUES ?classImportance { <classSimilarityWeight> }
            BIND(?classImportance AS ?importance)
            BIND(( 1 / 3 ) AS ?levelImportance)
            ?similarityConfiguration
                    rs:hasPropertySimilarity  ?propertySimilarity .
            ?propertySimilarity
                    rs:appliedOnProperty  ?property .
            ?similarityConfiguration
                    rs:hasClassSimilarity  ?classSimilarity .
            ?classSimilarity
                    rs:appliedOnClass     ?targetClass ;
                    rs:hasClassSimilarityValue  ?similarity .
            ?x      ?property          ?y .
            ?item   ?property          ?z .
            ?y      rdf:type           ?targetClass .
            ?z      rdf:type           ?targetClass
            FILTER ( ?y != ?z )
            BIND(concat("both of ", strafter(str(?y), "#"), " and ", strafter(str(?z), "#"), "
are from type ", strafter(str(?targetClass), "#")) AS ?becauseOf)
          }
        FILTER ( ?x != ?item )
```

```
      }
    OPTIONAL
      {

        ?userContext  rdf:type       rs:UserContext ;
                rs:appliedOnItems  ?itemClass ;
                rs:appliedOnUsers  ?userClass
        OPTIONAL
          { ?userContext  rs:hasWeightIfContextMatched  ?weightMatched }
        OPTIONAL
          { ?userContext  rs:hasWeightIfContextDoesNotMatch  ?weightNotMatched }
        OPTIONAL
          { ?userContext  rs:doNotRecommendInCaseNotMatch  true
            BIND(1 AS ?skip_)
          }

      }

      VALUES ?user { <USERURI> }
      bind(if (bound(?skip_) && (not EXISTS {?user a ?userClass}) && (EXISTS
{?item a ?itemClass}), ?skip_, 0) as ?skip1UserContext)



      values (?defaultUserMatched ?defaultUserNotMatched) {(<defaultUserMatched>
<defaultUserNotMatched> )}
        BIND(if(EXISTS { ?user  rdf:type  ?userClass }, coalesce(?weightMatched,
?defaultUserMatched), coalesce(?weightNotMatched, ?defaultUserNotMatched)) AS
?weight)




    bind (if ( exists {?item a ?itemClass }, true , false) as
?doesItemBelongToUserContextItemClass)
      values ?defaultNoUserContext {<defaultNoUserContext>}
```

```
BIND(if(bound(?skip1UserContext), ?skip1UserContext, 0) as ?skipUserContext)
BIND( if ( !?doesItemBelongToUserContextItemClass , ?defaultNoUserContext
,if(bound(?weight), ?weight, ?defaultNoUserContext)) AS ?userContextWeight)



#OPTIONAL
#{
  VALUES ( ?defaultMatchedTemporalContext
?defaultNotMatchedTemporalContext ?defaultNoTemporalContext ) {
    ( <defaultMatchedTemporalContext> <defaultNotMatchedTemporalContext>
<defaultNoTemporalContext> )
    }
    ?temporalContext
        rdf:type rs:TemporalContext ;
        rs:appliedOnItems ?itemClassTemporal .
        OPTIONAL{ #new
    ?item rdf:type ?itemClassTemporal
    OPTIONAL
      { ?temporalContext
          rs:canBeRecommendedFrom ?fromLimit
      }
    OPTIONAL
      { ?temporalContext
          rs:canBeRecommendedUntil ?untilLimit
      }
    OPTIONAL
      { ?temporalContext
          rs:hasWeightIfContextMatched ?matchedWeight
      }
    OPTIONAL
      { ?temporalContext
          rs:hasWeightIfContextDoesNotMatch ?unmatchedWeight
      }
```

```
                OPTIONAL
                  { ?temporalContext
                          rs:doNotRecommendInCaseNotMatch  true
                   BIND(1 AS ?skip_)
                  }
                BIND(   if(((bound(?fromLimit))   &&   (   now()   <   ?fromLimit   )   &&
(bound(?skip_)))
              || (bound(?untilLimit) && (now() > ?untilLimit) && bound(?skip_))
              , ?skip_, 0) as ?skip1TemporalContext)

                BIND(if((  bound(?fromLimit)  &&  bound(?untilLimit)  ),  if((  now()  <
?fromLimit  ),  coalesce(?unmatchedWeight,  ?defaultNotMatchedTemporalContext),  if((  (
?fromLimit  <=  now()  )  &&  (  now()  <=  ?untilLimit  )  ),  coalesce(?matchedWeight,
?defaultMatchedTemporalContext),                               coalesce(?unmatchedWeight,
?defaultNotMatchedTemporalContext))),  if(bound(?fromLimit),  if((  now()  <  ?fromLimit  ),
coalesce(?unmatchedWeight,                          ?defaultNotMatchedTemporalContext),
coalesce(?matchedWeight,  ?defaultMatchedTemporalContext)),  if(bound(?untilLimit),  if((
?untilLimit  <  now()  ),  coalesce(?unmatchedWeight,  ?defaultNotMatchedTemporalContext),
coalesce(?matchedWeight,                              ?defaultMatchedTemporalContext)),
?defaultNoTemporalContext))) AS ?temporalWeightHelper)
                }
          bind     (     if     (bound(?temporalWeightHelper)     ,     ?temporalWeightHelper,
?defaultNoTemporalContext) as ?temporalWeight)
               #}
              VALUES ?defaultNoTemporalContext2 { <defaultNoTemporalContext> }
              BIND(if(bound(?skip1TemporalContext),    ?skip1TemporalContext,    0)    AS
?skipTemporalContext)
              BIND(if(bound(?temporalWeight),                         ?temporalWeight,
?defaultNoTemporalContext2) AS ?temporalContextWeight)


         }



          }
```

```
GROUP BY ?item

HAVING (( SUM(if(?skipTemporalContext, 1, 0)) = 0 ) && ( SUM(if(?skipUserContext, 1, 0)) = 0 ))
ORDER BY DESC(?recommendedItemSimilarity)
limit <numOfMaxRecommendedItems>
```

10.5 Available Recommendation Services

10.5.1 Level 0 Instance Recommendation Service

The final user can request this server from the Web Interface component; tab "*Level 0 Instance*", as illustrated in Figure 10-3. The description of the fields is as bellow:

- *USER URI* field[1]: it represents the user's URI that the system should generate recommendation for.
- *INSTANCE SIMILARITY WEIGHT* field: it represents the value of the variable Level0Importance
- *NUMBER OF MAXIMUM RECOMMENDED ITEMS* field[2]: it represents the maximum number of items that the Recommender System should generate.

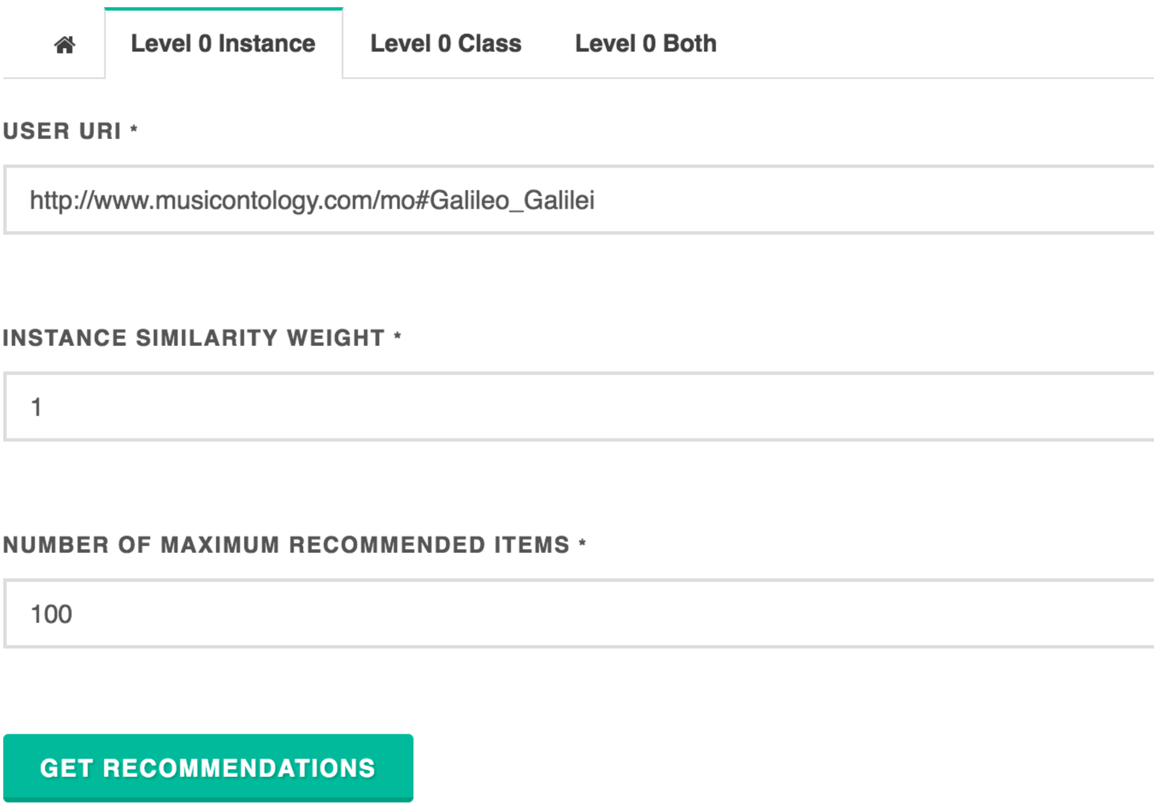


Figure 10-3 Level 0 Instance Recommendation Service

When the user clicks on *GET RECOMMENDATION* button, a GET HTTP request is being generated and sent to the *Web Service* component, which will extract the parameters'

[1] USER URI field exists in all the services. It has the same meaning in all of them, so we do not need to repeat its description for each service.

[2] NUMBER OF MAXIMUM RECOMMENDED ITEMS exists in all the service. It has the same meaning in all of them, so we don't need to repeat its description for each service.

values and calls the related SPARQL Interface API, which will load the corresponding SPARQL template and customize it according to the received parameters' values [1].

The query template's file name is level0instanceSimilarityForUser.rq exits in the provided code. Looking at the query, we can see the triple illustrated in Table 10-9, where the meaning of the prefix is illustrated in Table 10-8 [2]. That triple is an inferred triple, which means we need an OWL reasoner to generate that triple in the RDF Store. In case of turning off the reasoner, we need to customize the query to select what are the rating classes that are considered subclasses of the *rs:Likes* class, as described in more details in section 0

Table 10-8 SPARQL Prefix meanings

Prefix	URI	Meaning
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	RDF vocabularies
owl	http://www.w3.org/2002/07/owl#	OWL vocabularies
rdfs	http://www.w3.org/2000/01/rdf-schema#	RDFs vocabularies
xsd	http://www.w3.org/2001/XMLSchema#	XSD vocabularies
rs	http://www.SemanticRecommender.com/rs#	Recommender System Ontology vocabularies
mo	http://www.musicontology.com/mo#	Domain Ontology vocabularies
:	http://www.musicsemanticontology.com/mso#	Joined Ontology vocabularies

Table 10-9 Inferred Triple Liked Ratings

Subject	Predicate	Object
?ratings	Rdf:type	Rs:Likes

10.5.2 Level 0 Class Recommendation Service

The final user can request this server from the Web Interface component; tab "Level 0 Class", as illustrated in Figure 10-4. The description of the fields is as bellow:

[1] This flow is the same for all Recommendation Services, so we won't repeat it later. We will just describe and discuss the generated SPARQL query.

[2] We will consider the same meanings for the same prefix in this chapter.

- *CLASS SIMILARITY WEIGHT* field: it represents the value of the variable Level0Importance

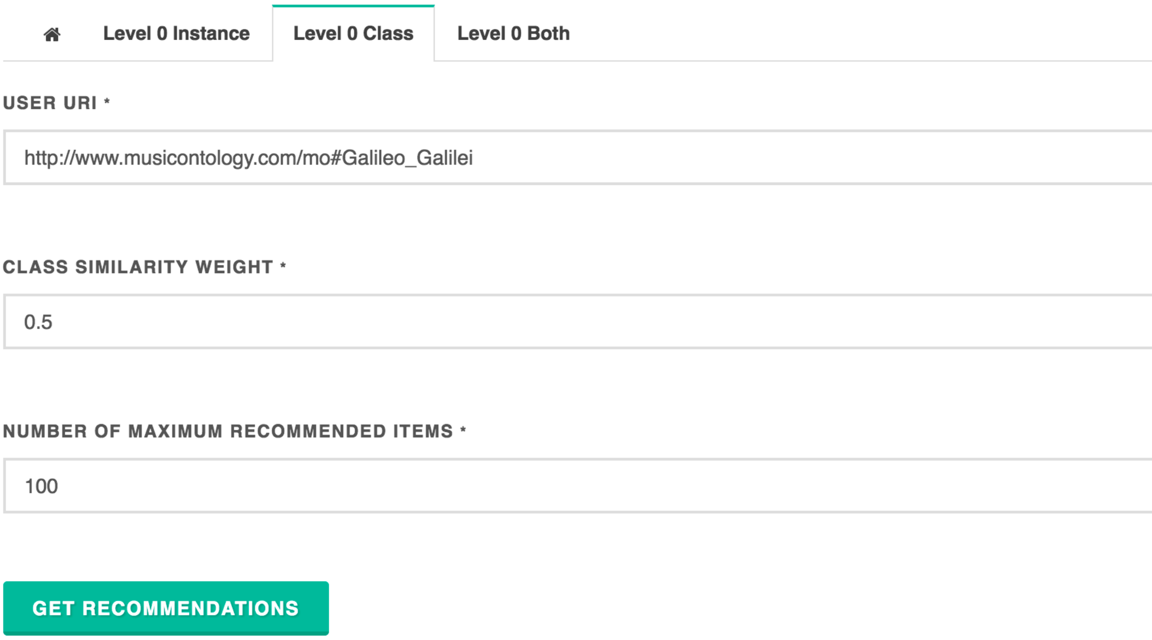


Figure 10-4 Level 0 Class Recommendation Service

10.5.3 Level 0 Instance and Class Recommendation Service

The final user can request this server from the Web Interface component; tab "Level 0 Both", as illustrated in Figure 10-5.

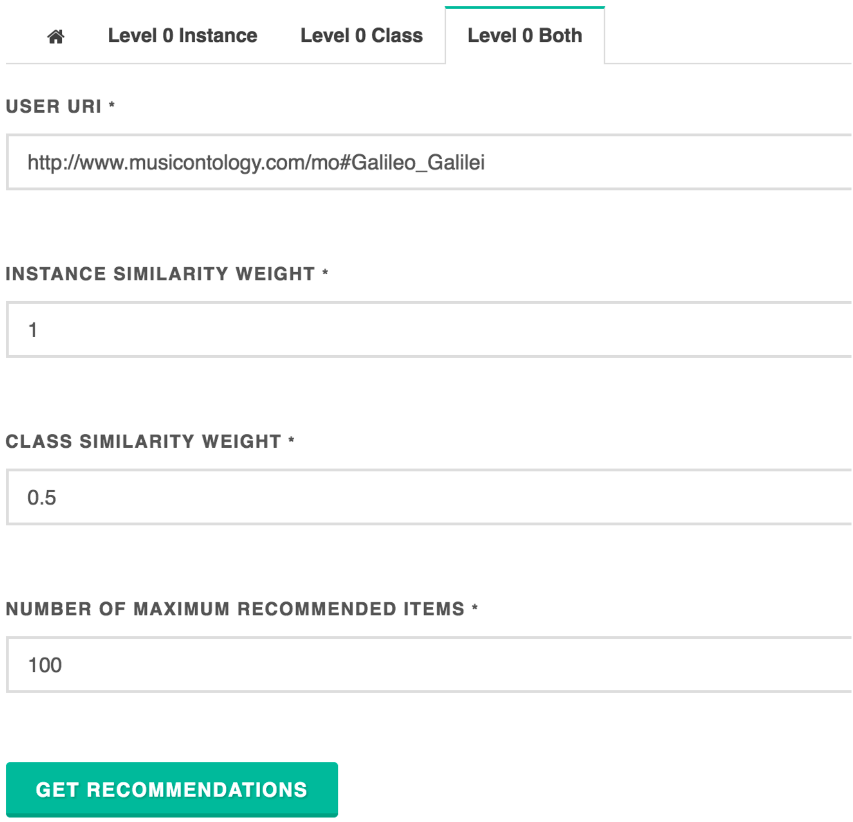


Figure 10-5 Level 0 Instance and Class Recommendation Service

10.5.4 Level 1 Instance Recommendation Service

The final user can request this server from the Web Interface component; tab "Level 1 Instance Recommendation Service", as illustrated in Figure 10-6.

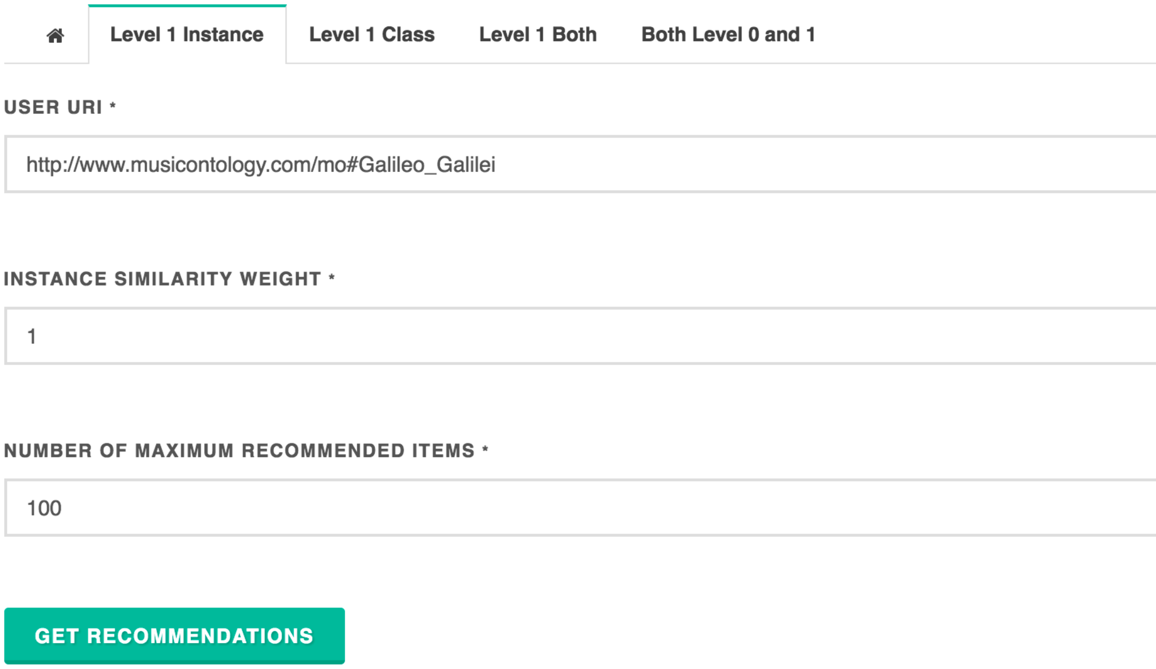


Figure 10-6 Level 1 Instance Recommendation Service

10.5.5 Level 1 Class Recommendation Service

The final user can request this server from the Web Interface component; tab "Level 1 Class", as illustrated in Figure 10-7.

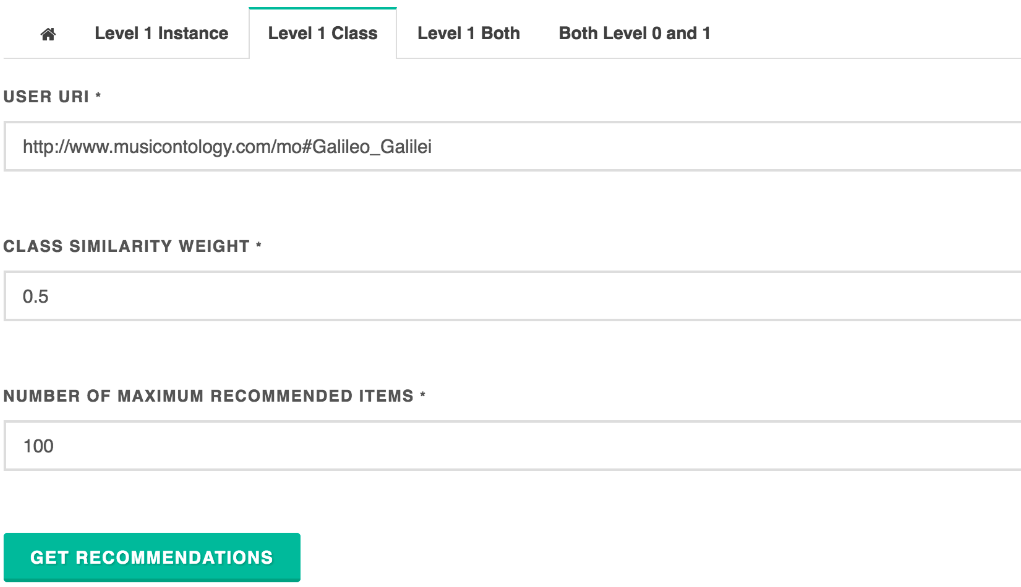


Figure 10-7 Level 1 Class Recommendation Service

10.5.6 Level 1 Instance and Class Recommendation Service

The final user can request this server from the Web Interface component; tab "Level 1 Both", as illustrated in Figure 10-8

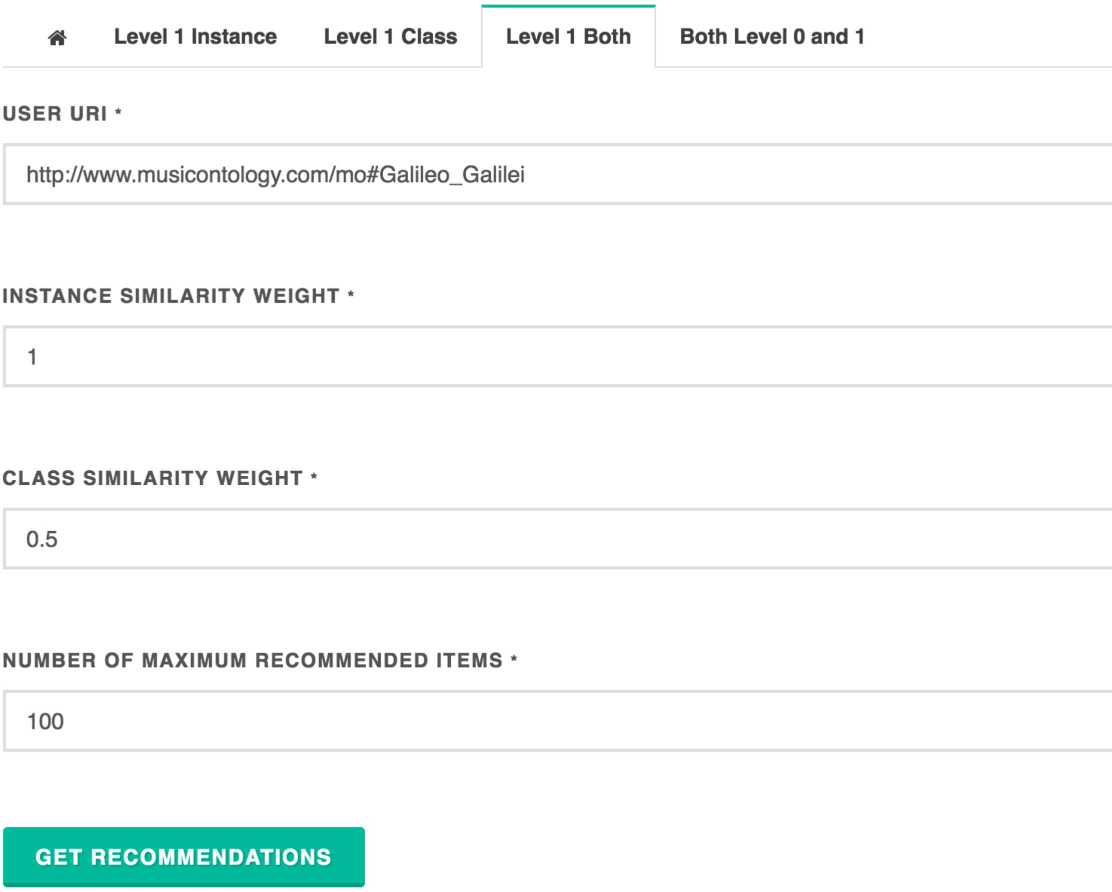


Figure 10-8 Level 1 Instance and Class Recommendation Service

10.5.7 Level 1 and Level 0 Recommendation Service

The final user can request this server from the Web Interface component; tab “Both Level 0 and 1”, as illustrated in Figure 10-9

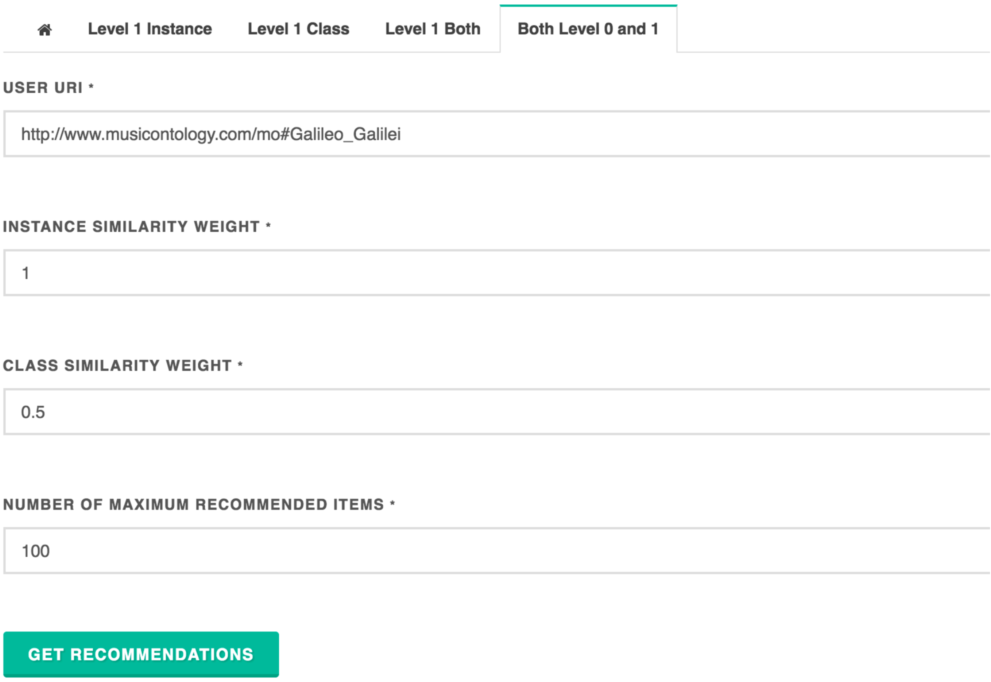


Figure 10-9 Level 0 and 1 Recommendation Service

10.5.8 User Context Recommendation Service

The final user can request this server from the Web Interface component; tab "User Context", as illustrated in Figure 10-10. The descriptions of the fields are as bellow:

- *DEFAULT USER MATCHED* field: it represents the default value of the UserContextWeight, which is illustrated in section 7.7.13, if both the user is from the type specified by the UserContext user class, and the item is from the type specified the UserContext item class.
- *DEFAULT USER NOT MATCHED* field: it represents the default value of the UserContextWeight, which is illustrated in section 7.7.13, if the user is not from the type specified by the UserContext user class, and the item is from the type specified by the UserContext item class.
- *DEFAULT NO USER CONTEXT* field: it represents the default value of the UserContextWeight, which is illustrated in section 7.7.13, if the item does not belong to the type specified by the UserContext item class.

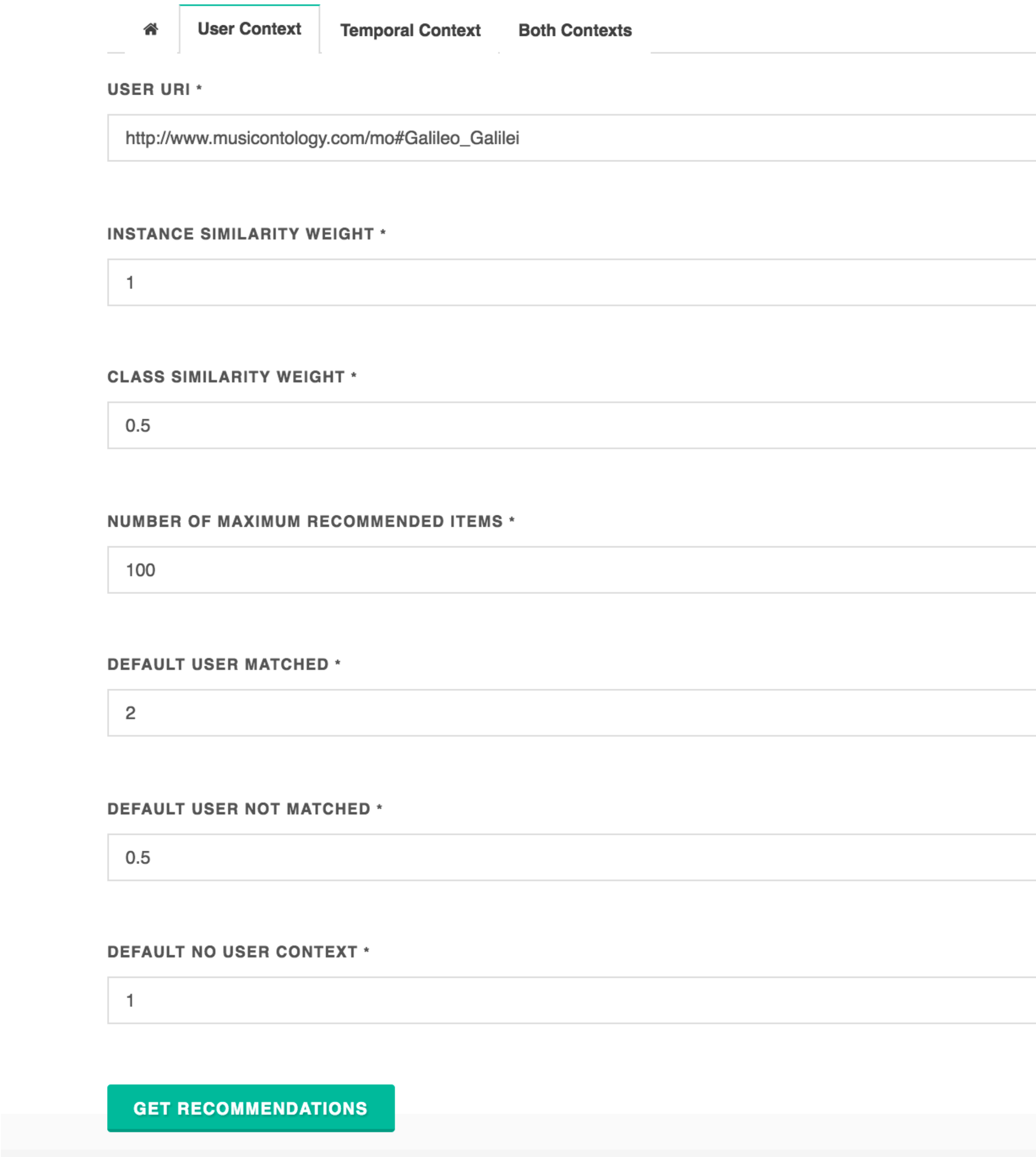


Figure 10-10 User Context Recommendation Service

10.5.9 Temporal Context Recommendation Service

The final user can request this server from the Web Interface component; tab "Temporal Context", as illustrated in Figure 10-11. The descriptions of the fields are as bellow:

- *DEFAULT MATCHED TEMPORAL CONTEXT* field: it represents the default value of the *TemporalContextWeight*, which is illustrated in section 7.7.15, if the item is from the type specified by the *TemporalContext* item class, and the time of generating the recommendations is within the time specified by the *TemporalContext* instance.

- *DEFATUL NOT MATCHED TEMPORAL CONTEXT* field: it represents the default value of the *TemporalContextWeight*, which is illustrated in section 7.7.15, if the item is from the type specified by the *TemporalContext* item class, and the time of generating the recommendations is not within the time specified by the *TemporalContext* instance.
- *DEFAULT NO TEMPORAL CONTEXT* field: it represents the default value of the TemporalContextWeight, which is illustrated in section 7.7.15, if the item is not from the type specified by the *TemporalContext* item class.

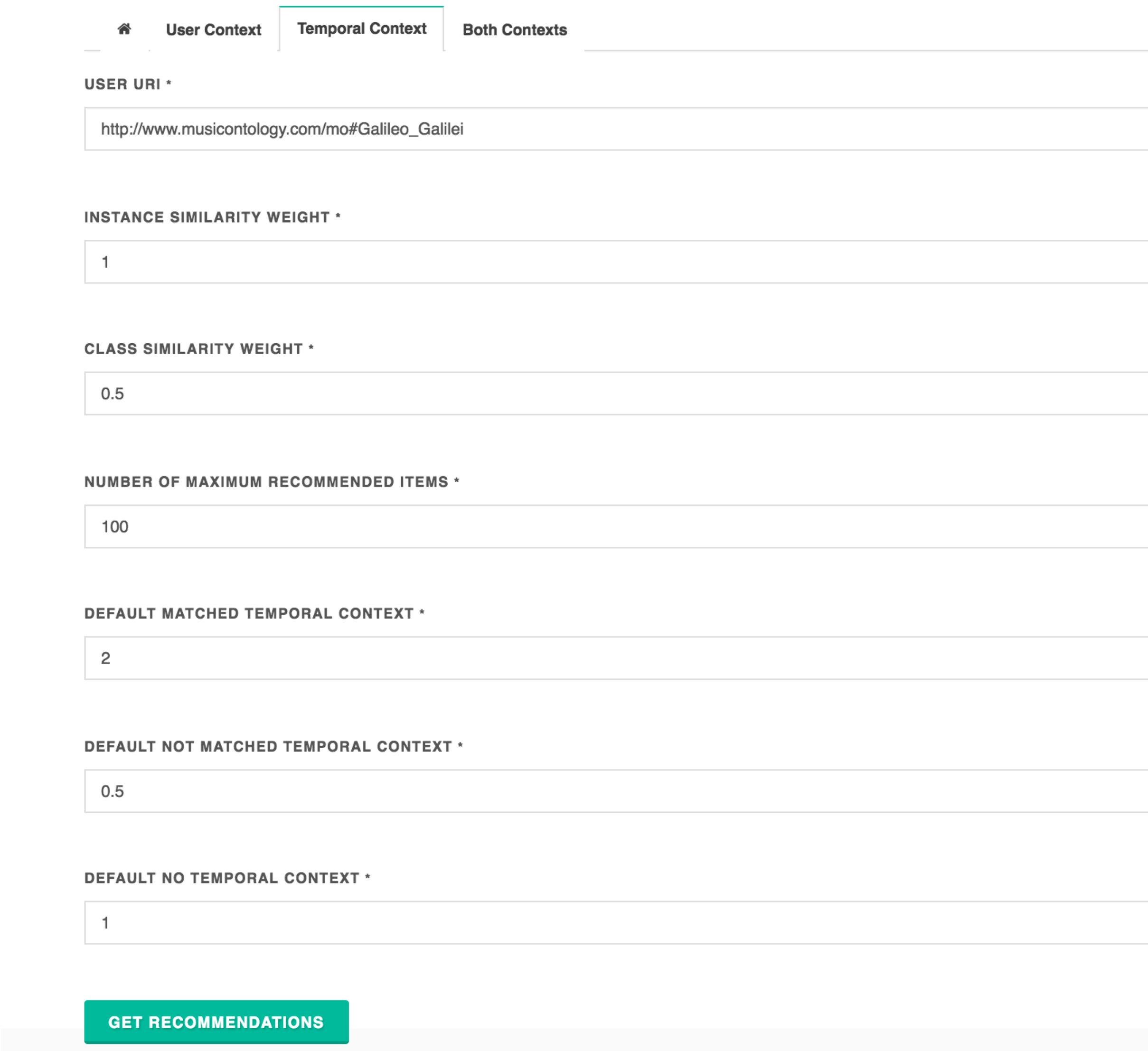


Figure 10-11 Temporal Context Recommendation Service

10.5.10User and Temporal Contexts Recommendation Service

The final user can request this server from the Web Interface component; tab "Both Contexts", as illustrated in Figure 10-12

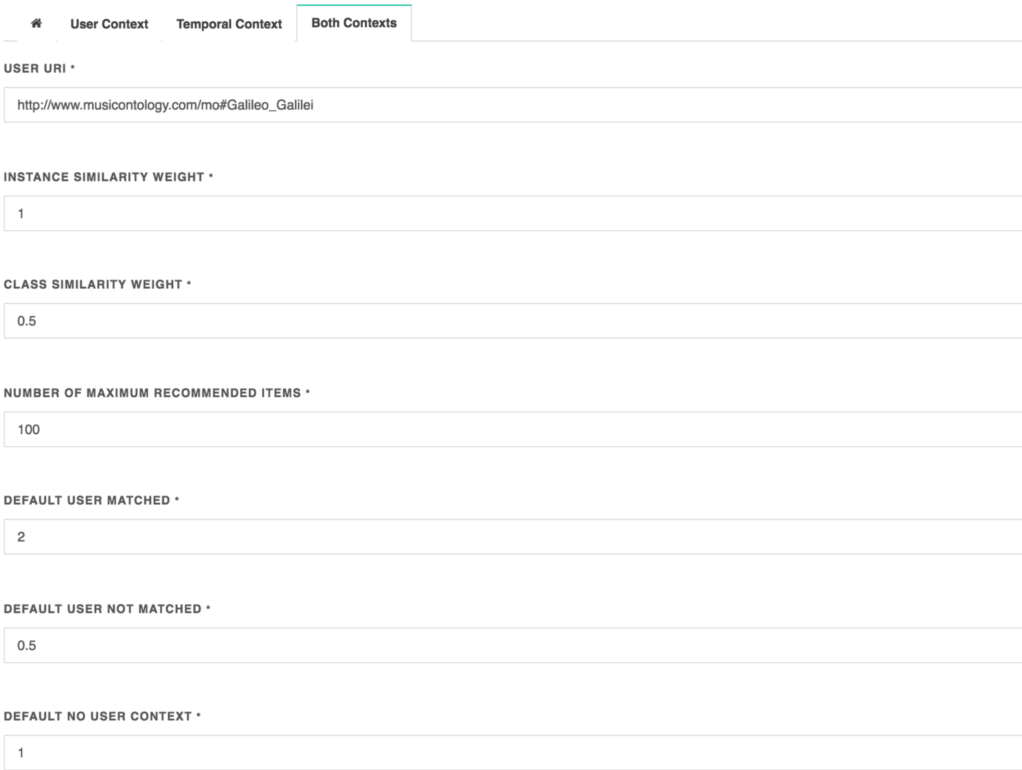


Figure 10-12 User and Temporal Contexts Recommendation Service

10.5.11Countable Recommendation Service

The final user can request this server from the Web Interface component; tab "*Countable All*", as illustrated in Figure 10-13. The descriptions of the fields are as bellow:

- *NUMBER OF COUNTABLE* field: it represents the maximum number of countable values that the system will suggest items related to them, as described in section 7.7.20.
- *THREE SHOULD FOR AVERAGE* field: it represents the minimum accepted average of ratings for the items related to the countable value, as illustrated in section 7.7.20

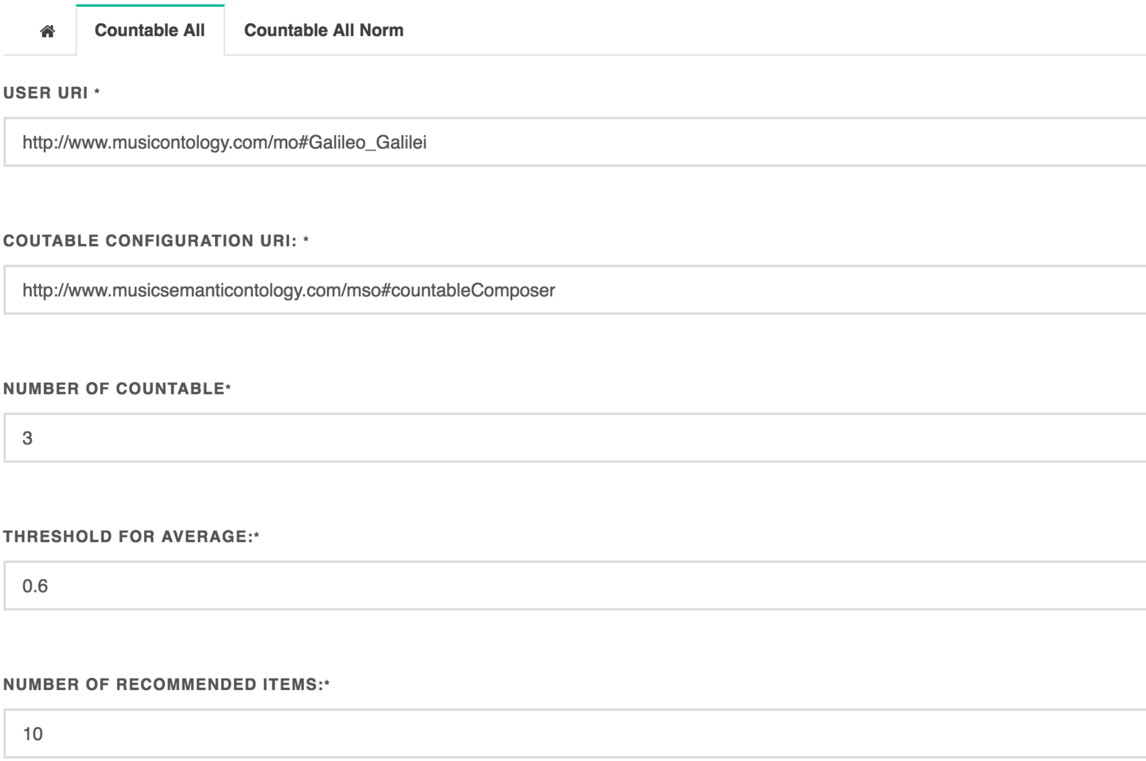


Figure 10-13 Countable Recommendation Service

10.5.12Countable Norm Recommendation Service

The final user can request this server from the Web Interface component; tab "*Countable All Norm*", as illustrated in Figure 10-14. The descriptions of the fields is as bellow:

- *NUMBER OF ITEM FOR EACH COUNTABLE VALUE* field: it represents maximum number of recommended items for each countable value, as illustrated in section 7.7.21.

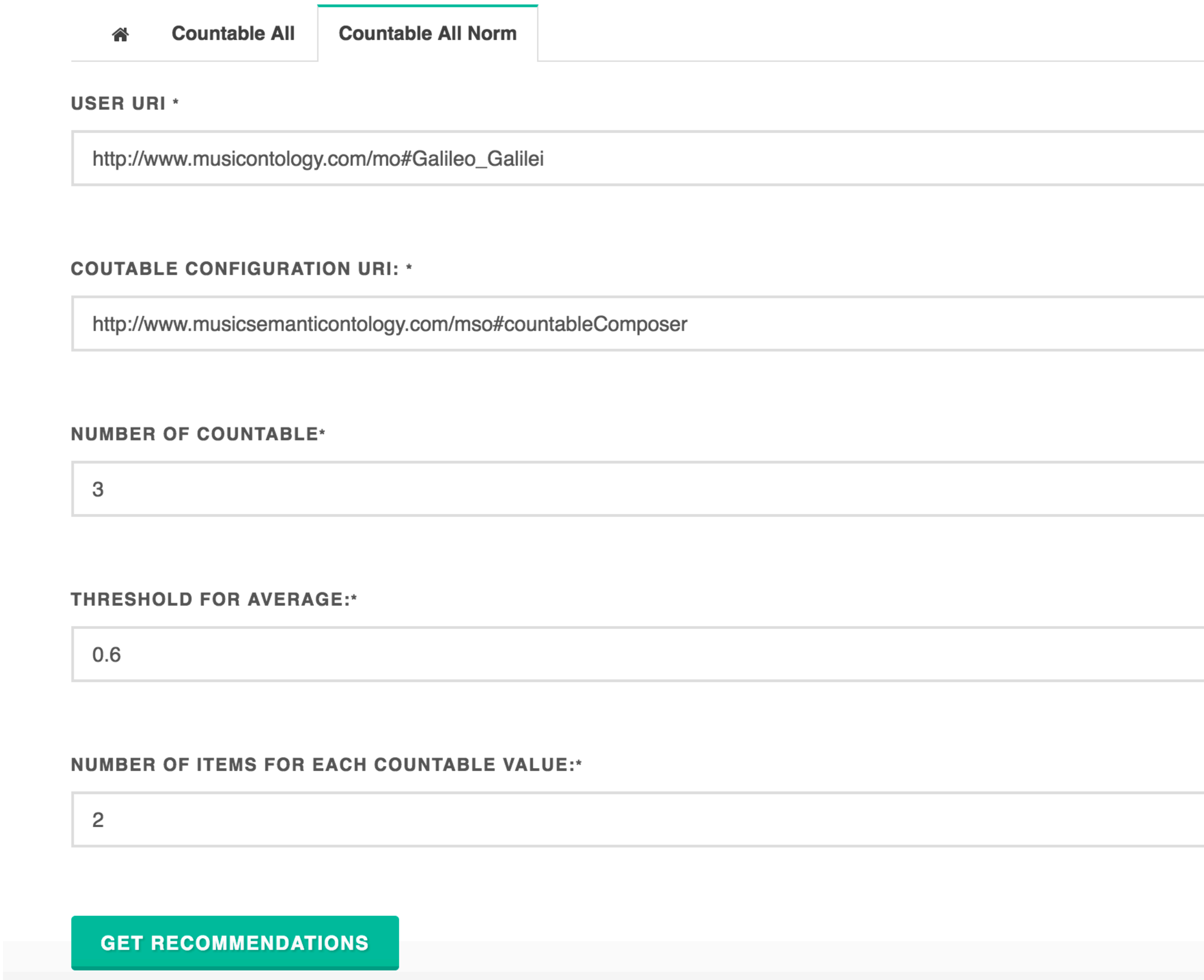


Figure 10-14 Countable Norm Recommendation Service