Dipartimento di Ingegneria e Scienza dell'Informazione

KDI Lab 2019-20

smartRemedy - A support for basic healthcare needs

Document data:
14.12.2019

Reference persons:

Informal-to-Formal subgroup
Martina Maffei - 205144
Matteo Proch - 203435
Riccardo Scandino* - 203421
Stefania Mattevi - 207651

Data Integration subgroup
Alberto Bellumat - 213898
Matteo Zanoni - 203393
Roberto Negro** - 211505

**subgroup leader*
***subgroup and project leader*

Index

Revision History

Revision	Date	Description of Changes
0	04.10.2019	Report initialization, scenario definition, personas drafted
1	10.10.2019	Scenario reviewed, personas reviewed
2	17.10.2019	Competences queries drafted, database selection section drafted
3	24.10.2019	Queries reviewed, data selection reviewed and scraping of data drafted
4	31.10.2019	First ER attempt, scraping of the data section finalized
5	07.11.2019	Model design finalized, data preparation section drafted
6	14.11.2019	Protégé modelling section drafted
7	21.11.2019	Formalization section finalized, data preparation section finished
8	28.11.2019	Integration with Karma, fixes on previous sections, top-level grounding and lexical information sections
9	05.12.2019	Data integration section finalized, Knowledge graph description section with GraphDB queries
10	12.12.2019	Small edits and fixes
11	14.12.2019	Finalization of the report

1. Scenario Description

Our project aims to provide a tool that supports consumers in their basic health needs, from the discovery of symptom to the buying of an appropriate drug according to their needs. It gives an exhaustive list of medicines related to the active principles that best asses the symptom provided by the user. Once the product has been selected, it's possible to obtain all the information about it (e.g. its administration route, intake form and side effects), it is also possible to decide whether to buy it online or in a drugstore. In addition, our application finds the pharmacy's location, its opening hours, and many other details connected to that store.

Now we show a series of simple possible scenarios in which our application can be used.

In the first scenario, Mr. Abraham is a 70 years old English gentleman from London who has the common diseases of people of his age and even some allergies. One morning, he woke up with some back pain, and he decided to look for the right medicine to reduce his pain in his medicines shelf. There was a mess in it, and Abraham had no clue on which was the correct drug. He found a drug similar to the one he used to use but the package leaflet was not present, and therefore he refused to assume it.

In another scenario, Mrs. Marge is a 45 years old American divorced housewife with three little children: Maggie, Bart, and Lisa. They are 14, 11, and 5 years old respectively. She is on a very tight budget so she's trying to save as much money as possible to pay her bills. One day coming home from school, Maggie had a terrible cough and Marge wanted to give her some medicine. At home, the only drug for this symptom was in pills but Maggie was not able to swallow them and refused to take them. Marge was desperate, and she didn't know what to do.

In the last scenario, Mr. Homer is a 29 years old Australian businessman on a work trip in Trento. It was a rainy day, he was stressed out by his job and this caused him a severe headache. He had no medicine with him; hence, he asked a local for the closest drugstore. After a long walk, he found out that it was closed. He needed a pharmacy near his position to buy some medicine, but he didn't know how to deal with this problem.

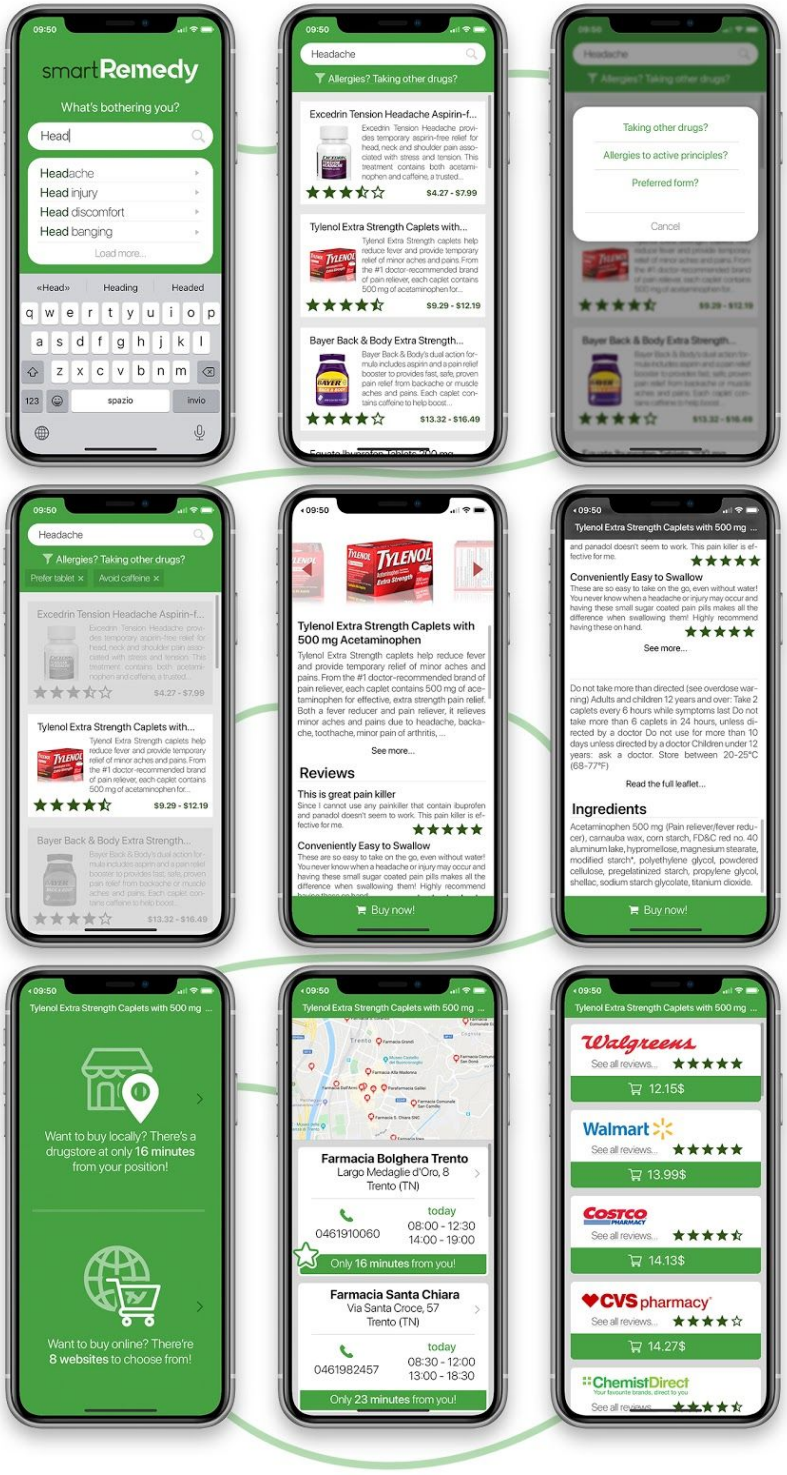


Figure 1. Mockup of a possible real-world application of smartRemedy.

1.1 Storytelling definition

Persona	Real-world action by persona	System/demo action
Mr. Abraham	Searches the drug Voltadvance that he found in his medicines shelf	Returns all the information about Voltadvance (intake form, dosage form, warnings, ..)
Mr. Abraham	Searches drugs for the back pain	Returns all the possible drugs for the back pain
Mr. Abraham	Searches the symptoms cured by Aspirin	Returns a list of all symptoms cured by Aspirin
Mr. Abraham	Searches the active principles cured by Aspirin	Returns a list of all active principles cured by Aspirin
Mr. Abraham	Searches the drugs without paracetamol to solve back pain	Returns all the possible drugs that do not contain paracetamol to solve back pain
Mr. Abraham	Searches all the drugs for back pain with oral administration	Returns all the drugs for back pain with oral administration
Mr. Abraham	Searches all the drugs for back pain with oral administration but that do not contain paracetamol	Returns all the drugs for back pain with oral administration that do not contain paracetamol
Mr. Abraham	Searches all the drugs without paracetamol that are not with oral administration	Returns all the drugs without paracetamol with all the possible administration routes except the oral one
Mr. Abraham	Searches the picture of the package of Aspirin	Returns the picture of the package of Aspirin
Mr. Abraham	Searches the drugstores with wheelchair accessibility	Returns a list of all the drugstores with wheelchair accessibility
Mr. Abraham	Searches if Aspirin can be taken together with Moment	Returns a yes/no answer
Mr. Abraham	Searches if Aspirin is in pills	Returns a yes/no answer

Mr. Abraham	Searches alternative drugs to Aspirin that solve the same symptom	Returns a list of drugs that solve the same symptom of Aspirin
Mr. Abraham	Searches if Aspirin contains paracetamol	Returns a yes/no answer
Mr. Abraham	Searches if Aspirin can cure back pain	Returns a yes/no answer
Mr. Abraham	Searches the Aspirin description	Returns the Aspirin description
Mr. Abraham	Searches the usage dose of Aspirin	Return the usage dose information of Aspirin
Marge	Searches a drug for Maggie's cough	Returns a list of drugs to solve the cough
Marge	Searches a drug for Maggie's cough buyable online	Returns a list of drugs to solve cough with the option to buy it online
Marge	Searches if Bisolvon is buyable online	Returns a yes/no answer. In case of negative response, it shows a list of cough's drugs buyable online
Marge	Searches the drugs for the cough without a doctor's prescription	Returns a list of drugs to solve cough without the need of the doctor's prescription
Marge	Searches drugs for the cough that are not in pills	Returns a list of drugs for the cough that are not in pills
Marge	Searches all the drugs for the cough ordered by price	Returns all the drugs for cough ordered by price
Marge	Searches all the drugs for the cough ordered by rating	Returns all the drugs for cough ordered by rating
Marge	Searches all the drugs for the cough that cost less than 20€ ordered by rating	Returns a list of drugs for cough that cost less than 20€ ordered by rating
Marge	Searches the best drug for the cough that costs less than 20€	Returns the best drug for the cough that costs less than 20€

Marge	Searches the reviews of Bisolvon	Returns a list with all the reviews of Bisolvon
Marge	Searches the drugstores with parking lots	Returns a list of drugstores with parking lots
Marge	Searches the rating of Bisolvon	Returns the rating and the reviews of Bisolvon
Marge	Searches on which website she can buy Bisolvon	Returns a lists of websites where to buy Bisolvon
Marge	Searches the average delivery time of the website where she bought Bisolvon	Returns the average delivery time of the website where she bought Bisolvon
Marge	Searches the payments accepted by the website where she bought Bisolvon	Returns all the payments accepted by the website where she bought Bisolvon
Marge	Searches the price of Bisolvon	Returns the Bisolvon price
Marge	Searches all the drugs for the cough that costs less than 20 €	Returns all the drugs for cough that costs less than 20 euro
Marge	Searches the drugs for the cough without a doctor's prescription that cost less than 20€	Returns a list of drugs for the cough without a doctor's prescription that cost less than 20€
Homer	Searches all the drugstores near its position	Returns a list of drugstores near its position
Homer	Searches all the drugstore open now	Returns a list of open drugstores
Homer	Searches if there are drugstores in Trento	Returns a yes/no answer
Homer	Searches all the drugstores in Trento	Returns a list of all the drugstores in Trento
Homer	Searches 'Farmacia Gallo' drugstore	Returns a list with: opening hours, position and other info of 'Farmacia Gallo'

Homer	Searches the telephone number of the drugstore ‘Farmacia Gallo’	Returns the telephone number of ‘Farmacia Gallo’
Homer	Searches the address of the drugstore ‘Farmacia Gallo’	Returns the address of ‘Farmacia Gallo’
Homer	Searches the photo of ‘Farmacia Gallo’	Return all photos of ‘Farmacia Gallo’
Homer	Searches if at ‘Farmacia Gallo’ is available the POS payment option	Returns yes/no answer
Homer	Searches if ‘Farmacia Gallo’ is opened now	Return yes/no answer
Homer	Searches if ‘Farmacia Gallo’ accepts cash	Returns a yes/no answer
Homer	Searches if ‘Farmacia Gallo’ accepts American dollar	Returns a yes/no answer
Homer	Searches headache drugs	Returns a list of headache drugs
Homer	Searches the headache drug with the best rating	Return the drug for headache with the best rating
Homer	Searches headache drugs ordered by rating	Return the drug for headache ordered by rating
Homer	Searches if Aspirin requires a doctor’s prescription	Returns a yes/no answer
Homer	Searches headache drugs without a doctor’s prescription	Returns a list of headache drugs without a doctor’s prescription
Homer	Searches the price of Aspirin	Returns the Aspirin price

1.2. Personas Description

As we deduced from the previous scenarios, we have three types of personas. Now, we see them in details.

Elderly person with allergy problems. Mr. Abraham is 70 years old. He is interested in *SmartRemedy* because he wants to search for details on specific drugs he has at home. The package leaflet is unavailable for the majority of its drugs, and he does not know what medicine to use to solve his back pain. Also, he is allergic to paracetamol, so he must be sure that what is taking will not contain it.
The part of the system that interests mr. Abraham most is the one connected to the details of the drug. By inserting the name of a drug in the application, a person can get all the details about it: intake quantity, assumption modality, dosage form, if it requires the prescription, active principle contained, usage dose, warnings and a brief description. If a person has allergy problems, he needs to pay attention if a medicine contains that specific active principle or not. For this purpose, *SmartRemedy* allows to search for medicines to solve a symptom based on some filters given by the user (for example, products that do not contain paracetamol).

Limited budget person. Marge is 45 years old and is a divorced housewife with three little children. She needs to save as much money as possible. She is interested in *SmartRemedy* because her children often get sick. In particular, Maggie has a cough, and the only drug at home for this symptom is in pills. The daughter has problems swallowing it, so Marge needs another medicine with different intake method. She uses the *SmartRemedy* to search remedies in a format different from pills within a budget of 20€. The part of the system she is interested in most is the one connected to the website and the reviews. The application allows her to buy products directly online and to see the other users' reviews. By inserting the symptom with some constraints (price range, sort the results by a specific category, and so on), the system returns a list of drugs. After selecting a product, SartRemedy gives the website with the best price from which to buy it. An important feature is that a drug can request the doctor's prescription: only the products without prescription can be bought online.

Foreign travelling person. Homer is an Australian businessman aged 29 on a work trip to Trento. He doesn't speak Italian and he has no cash with him. After asking a local person a drugstore location, he found it closed. He stays a few days in Trento so he cannot buy the product online and, for that reason, he is interested in our application to search open pharmacies close to his position. *SmartRemedy* gives him all the drugstores sorted by the proximity of its position. By selecting a result from the list, the system gives details on that specific store: name, telephone number, if it has wheelchair accessibility, if it has POS payment modality available, if there is a car park nearby and the opening hours. The 'search drugstore' section is the most useful for mr. Homer. There are three modalities to look for a drugstore: by looking

at the actual position, by searching the specific drugstore or by looking at the opening hours.

As you can see we choose to identify just 3 different personas. According to what discussed during lectures we prefer to select three situations dealing with the same level of knowledge about drug (no technical part is considered) but at the same time, they should be as separated as possible.
A future extension of our scenario (and of course of the application) could be the consideration of other technical personas.

2. Generalized Queries for Persona

Persona	Generalized queries
Mr. Abraham	Give me all medicine with a specific administration route
Mr. Abraham	Give me all drugs to cure a specific symptom
Mr. Abraham	Give me all drugs with a specific form
Mr. Abraham	Give me usage dose of a specific drug
Mr. Abraham	Give me all drugs incompatible with a specific medicine
Mr. Abraham	Give me all drugs to cure this symptom not containing specific active principle
Mr. Abraham	Give me package pictures of a specific medicine
Mr. Abraham	Give me all drugs of a specific class
Mrs. Marge	Given a specific active principle give me the cheapest drug online
Mrs. Marge	Give me all drugs sold online
Mrs. Marge	Give me all shippable drugs
Mrs. Marge	Give me all warnings of a specific drug
Mrs. Marge	Give me all drugs to cure this symptom ordered by price
Mrs. Marge	Give me all drugs to cure this symptom ordered by rating
Mrs. Marge	Give me all websites with a specific payment method
Mrs. Marge	Give me all drugs that have prescription
Mrs. Marge	Give me all reviews of a specific drug

Mrs. Marge	Give me all reviews of a specific website
Mr. Homer	Tell me the nearest drugstore from here
Mr. Homer	Give me all drugstores open now
Mr. Homer	Give me drugstore with nightshift today
Mr. Homer	Give all drugstores with a specific payment method
Mr. Homer	Give me all the drugstores in a specific town
Mr. Homer	Give me all contact information of this drugstore
Mr. Homer	Give me all photos of a specific drugstore

3. From Informal to Formal

3.1. Model design

From the generalized queries, we start building the ER model. This process consists of extracting keywords from the generalized queries in order to identify the entities of our model and the relationships present between them. These entities are organized into three levels of decreasing importance: *core*, *accessory*, and *common* entities. The final ER model can be seen in section 3.4. Model Visualization, Figure 3.

Core Entities. These entities are the ones retrieve by a generalized query. Here we define the entities *Drug*, *DrugWithoutPrescription*, *Drugstore*, *WebSite*, *AggregateRating*:

Drug. This is the main entity, representing all the drugs with and without *prescription* available in the datasets. Each drug has a *name*. Among the available attributes of this entity to be highlighted are *administrationRoute*, describing the way by which a drug or agent enters the body, e.g. by mouth, etc.; *dosageForm,* is the form in which a drug is marketed for use, e.g. pill, powder, etc.; *description* and *warning,* containing the description of the drug, and all the warnings regarding it, such as collateral effects; *isAvailableGenerically* defining if the drug has a generic alternative or not. A drug contains active principles (*contains* relation with *ActiveIngredients*), it may not have a doctor's prescription (*hasPrescription* relation with *DrugWithoutPrescription*) and it has some dose schedule (*hasDose* relation with *MaximumDoseSchedule*). Furthermore, it has a relation with *AggregateRating* called *hasRating* in order to get the reviews of the medicine, and an *isSoldBy* relation with *Drugstore* to find the pharmacies selling the drug.

DrugWithoutPrescription. This entity inherits all the attributes from *Drug* (*isA* relation with *Drug* entity) with an important distinction: the attribute *hasPrescription* here as a specific value, equal to "False". This distinction is fundamental in order to relate the entity *WebSite* with this entity. This was the main difficulty encountered during the design of the model: *WebSite* sells all the drugs without prescription, however, the entity *Drug* contains both drugs with and without prescription.

Drugstore. Entity representing the physical place of drugstore. Specific attributes of this entity are *parking, disableAccess, description, websiteURL*. These

attributes report the presence of a parking place, an access for disabled people, a description of the pharmacy, and the correlated website. *DrugStore* is a place (*isA* relation with the entity *Place*) and is also a store (*isA* relation with the entity *Store*). Furthermore, it has a *hasSchedule* relation with *TimeSchedule* and a *sells* relation with *Drug*.

Website. Here we find websites selling drugs online, all the drugs sold by websites are without prescription. Website attributes include *url*, *shippingAverage*, *numberOfPhotos*. This entity has three relations: *sells* connected to *DrugWithoutPrescription*, *hasRating* connected to *AggregateRating* and *sellsAt* connected to the *Cost* entity.

AggregateRating. All the reviews of a particular item. The entities having reviews are *WebSite* and *Drug/DrugWithoutPrescription*. This entity returns all the information about the rating of an item. It has an *aggregationOf* relation with *Review* because it is composed of attributes computed on many reviews. In details, *AggregateRating* has the attributes: *ratingCount, reviewCount, ratingValue, averageRating, bestRating* and *worstRating*.

Auxiliary Entities. These entities are used to add further information to the core entities. Here, we define the entities *ActiveIngredient, MedicalSignOrSymptom, MaximumDoseSchedule, TimeSchedule* and *cost*:

ActiveIngredient. It has the attribute *name* of the active principle contained in the drug. An active ingredient may not be taken together with another one, therefore, it has a relation starting and ending in itself called *Incompatibility* . The active ingredient is contained in a drug (*isContained* relation) and it has a relation with *MedicalSignOrSymptom* because it cures one or more users' symptoms.

MedicalSignOrSymptom. It's the symptom of the user. For example, it can be a headache, back pain or anything else that can be cured by the active ingredient. For that reason, it has only an attribute *SignOrSymptom* and a relation with *ActiveIngredient*. It is connected to *ActiveIngredient* since it can be cured by a certain active principle (*isCured* relation).

MaximumDoseSchedule. It's the detailed dosage schedule of a specific drug. In particular, it has the attributes: *doseUnit, doseValue, frequency* and

targetPopulation. The last one indicates the typology of people that assume the drug (e.g. children, older people and so on).

TimeSchedule. It's the timetable of a specific drugstore. It has the attributes: *openingHours, date* and *formatDate*. This entity is useful to see whether a certain store is open or not given time or a day of the week.

Cost. It stores all the prices of a drug sold by a website. It has attributes: *value* and *currency*. This entity has been created to associate *Website* and *DrugWithoutPrescription* in order to allow users to obtain the price of a specific medicine from every website available. Indeed, the same drug can have different prizes depending on the website selling them.

Common Entities. These entities can be re-used across the domains, they share information among other entities. Here, we define the entities *Place*, *Store* and *Review*:

Place. This entity contains all the information about a physical place. In particular, it has the attributes: *address, city, country, longitude* and *latitude* which represent the position of the store; *faxNumber, telephoneNumber, email* that are the contact details; and, as additional information, *photo* that show how the store is. It has only a relation with *Drugstore* because a drugstore is situated in a specific place (*isA* relation).

Store. This entity is shared by *Drugstore* and *Website* because is composed by the payment method accepted by them (*isA* relations). It has two attributes: *paymentAccepted*, the type of payment accepted (e.g. cash, credit card) and *currenciesAccepted* such as euro, American dollar or any type of currency.

Review. This entity represents a single review. It is composed by *reviewTitle*, *reviewBody* and *reviewRating*. The title and the body are the comments written by the user, the rating is the level of appreciation of a particular product.

3.2. Model formalization description

We formalized our model with *Protégé* and we followed the structure created in our EER model. All the entities present in the EER are reported as classes, all the

attributes as data properties and all the relations as object properties. For the entities, and the consequent classes construction we took inspiration from the Schema.org ontology. We obtained 13 classes, 14 object properties and 73 data properties (Figure 2).

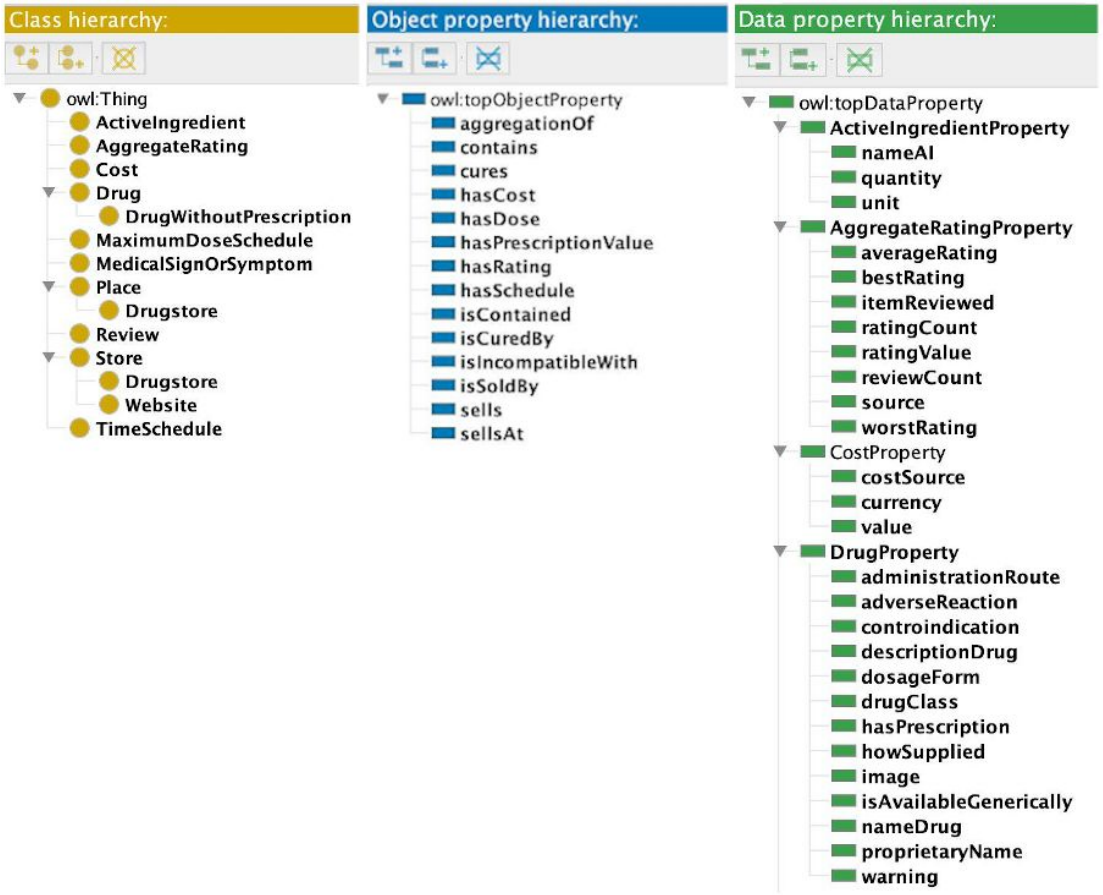


Figure 2. Our classes, object properties and data properties in Protégé. Data properties shown are a portion of the total.

During the implementation of the formal model and the integration with the datasets we found out some problem at the informal level. One example in the connection between drug and the website that initially were directly connected but, since it is possible to buy online only drugs that don't need a medical prescription, we created the intermediary *DrugWithoutPrescription*. A representation of the ontology can be seen at section 3.4 Model visualization, Figure 4.

Here follows some of the final classes and subclasses we created with the respective connections:

Drug. The main class. Its relations are: *contains* some *ActiveIngredient*, *isSoldBy* some *Store, hasRating* some *AggregateRating* and *hasDose* some *MaximumDoseSchedule*.

DrugWithoutPrescription. Subclass of Drug because it's a particular case of the respective main class: it has a prescription value equal to false. The medicines that do not need the doctor's prescription are sold at a certain price, therefore, a connection with the class Cost is required: *hasCost* some *Cost*.

Place. The physical position of an object. It is a super-class and it has *Drugstore* as a subclass.

Drugstore. Subclass of place. The reason for that choice is that the pharmacy is situated in a specific place (as we already saw in the EER model). It is important to notice that this sub-class is also a sub-class of *Store* (it's a type of store). The drugstore relations are: *hasSchedule s*ome *TimeSchedule*.

Store. Super-class that *sells* some *Drug*. It has two sub-classes: *drugStore* and *Website*.

Drugstore. Sub-class of store for the reasons explained before.

Website. Sub-class of store because it's a particular type of store: it sells products online. With respect to the drugstore, it *hasRating* some *AggregateRating, sellsAt* some *Cost*. An important difference with the drugstore is that a website can only sell drugs that do not require a medical prescription: *sells* only *DrugWithoutPrescription.*

ActiveIngredient *cures* some *MedicalSignOrSymptom, isContained some drug* and *isIncompatibleWith some ActiveIngredient.*

AggregateRating is a union of more ratings, therefore it is an *aggregationOf* some *Review*.

MedicalSignOrSymptom is the symptom of the user and *isCuredBy* some *AciveIngredient*.

Here, instead, follows the data properties:

ActiveIngredientProperty: *NameAI, quantity, unit*. They are all the properties of the Active Ingredients.

AggregateRatingProperty: *averageRating, bestRating, itemReviewed, ratingCount, ratingValue, reviewCount, source, worstRating.*

CostProperty: *costSource, currency, value.*

DrugProperty: *administrationRoute, adverseReaction, controindication, descriptionDrug, dosageForm, drugClass, hasPrescription, howSupplied, image, isAvailableGenerically, nameDrug, proprietaryName, warning*

DrugstoreProperty: *descriptionDrugstore, disableAccess, nameDrugstore, parking, websiteURL*

MaximumDosageScheduleProperty: *packageLeaflet*

MedicalSignOrSymptomProperty: *signOrSymptom*

PlaceProperty: *address, city, country, email, email, faxNumber, latitude, longitude, photo, telephoneNumber.*

ReviewProperty: *maxRating, minRating, ratingSystem, reviewAuthor, reviewBody, reviewRating, reviewTitle.*

StoreProperty: *currenciesAccepted, paymentAccepted*

TimeScheduleProperty: *date, formatDate, openingHours, timezone*

WebSiteProperty: *nameWebSite, numberOfPhotos, shippingAverageTime, url.*

The datatypes we used are: *xsd:boolean, xsd:double, xsd:float, xsd:int, xsd:integer, xsd:string*.

3.2.1. Lexical information upload description

Considering the large usability that our model could have dealing with healthcare we decided to lay the foundation for multilanguage availment of our application. To achieve this, according to what discussed during the lecture, we exploited WordNet: a simple lexical database. Once looked for a word it gives back all the possible synset (set of synonyms as lexicalized concepts). Selecting the right one gives you the possibility to correctly translate it in other supported idioms.
The results of our research on the database could be subdivided into three main groups. The easiest one related to the words that are in the database. In this situation,

the only difficulty is related to the correct detection of the synset that best fits our situation. They are:

> **DRUG** (30){03252323} <noun.artifact>[06] S: (n) drug#1 (drug%1:06:00::) (a substance that is used as a medicine or narcotic)
>
> **REVIEW** (2){06422034} <noun.communication>[10] S: (n) review#2 (review%1:10:01::), critique#1 (critique%1:10:00::), critical review#1 (critical_review%1:10:00::), review article#1 (review_article%1:10:00::) (an essay or article that gives a critical evaluation (as of a book or play))
>
> **RATING** (3){00876484} <noun.act>[04] S: (n) evaluation#1 (evaluation%1:04:00::), rating#2 (rating%1:04:00::) (act of ascertaining or fixing the value or worth of)
>
> **WEBSITE** {06370600} <noun.communication>[10] S: (n) web site#1 (web_site%1:10:00::), website#1 (website%1:10:00::), internet site#1 (internet_site%1:10:00::), site#3 (site%1:10:00::) (a computer connected to the internet that maintains a series of web pages on the World Wide Web) "the Israeli web site was damaged by hostile hackers"
>
> **COST** (75){13296870} <noun.possession>[21] S: (n) cost#1 (cost%1:21:00::) (the total spent for goods or services including money and time and labor)
>
> **STORE** (25){04209460} <noun.artifact>[06] S: (n) shop#1 (shop%1:06:00::), store#1 (store%1:06:00::) (a mercantile establishment for the retail sale of goods or services) "he bought it at a shop on Cape Cod"
>
> **DRUGSTORE** (6){03254045} <noun.artifact>[06] S: (n) drugstore#1 (drugstore%1:06:00::), apothecary's shop#1 (apothecary's_shop%1:06:00::), chemist's#1 (chemist's%1:06:00::), chemist's shop#1 (chemist's_shop%1:06:00::), pharmacy#2 (pharmacy%1:06:00::) (a retail shop where medicine and other articles are sold)

Problems start arising considering multiple words. These are not present in our database. In some case, we managed to find a single word having a synset that perfectly describes our concept. According to that, this situation is treated as the previous one. They are:

TIME SCHEDULE, looking for SCHEDULE

SCHEDULE (11){05919534} <noun.cognition>[09] S: (n) agenda#1 (agenda%1:09:00::), docket#2 (docket%1:09:00::), schedule#1 (schedule%1:09:00::) (a temporally organized plan for matters to be attended to)

MEDICAL SIGN OR SYMPTOM, looking for SYMPTOM

SYMPTOM (8){14323139} <noun.state>[26] S: (n) symptom#1 (symptom%1:26:00::) ((medicine) any sensation or change in bodily function that is experienced by a patient and is associated with a particular disease)

MAXIMUM DOSE SCHEDULE, looking for DOSAGE

DOSAGE (6){13794246} <noun.quantity>[23] S: (n) dose#2 (dose%1:23:00::), dosage#1 (dosage%1:23:00::) (the quantity of an active agent (substance or radiation) taken in or absorbed at any one time)

ACTIVE INGREDIENT, looking for INGREDIENT

INGREDIENT (3){03575860} <noun.artifact>[06] S: (n) ingredient#1 (ingredient%1:06:00::) (a component of a mixture or compound)

In one particular case, we could not find support on WordNet and we need to perform a sort of integration of the database. For the word DRUG WITHOUT PRESCRIPTION we started from the DRUG definition seen previously, we updated the "without prescription" part and we changed the ID.

DRUG WITHOUT PRESCRIPTION: (30){03252324} <noun.artifact>[06] S: (n) drugwithoutprescription#1 (drugwithoutprescription%1:06:00::) (a substance that is used as a medicine or narcotic that can be used and sold only under medical certification)

3.3. Top-level Grounding

During the construction of our model, we considered as top-level grounding *Schema.org*, in order to confer to the model an overall consistency and structure with respect to a CSK (Common Sense Knowledge) top-level ontology.

Schema.org was selected due to its better compatibility to our knowledge field. Indeed, other top-level ontologies such as *SUMO,* or *DBPedia* do not cover properly healthcare classes. Considering the main entity of our model, *Drug,* it has fundamental auxiliary entity: *MaximumDoseSchedule.* Classes like this or other medical-related ones regarding symptoms, usage dose, etc. are not present in those ontologies, making them unsatisfactory for our purpose. Using *Schema.org* we can build a consistent ontology based on an existing structured top-level ontology and at the same time expand this ontology to improve it. Protégé implementation of our ER model results in Figure 4.

Starting from the Owl just shown we imported the *Schema.org* "all-level" rdf in such a way to integrate the two models. Almost all the classes are inspired by *Schema.org* so it is possible to notice, in Figure 5, that there is an overlap between the two models (nodes represented in orange). The only differences are for *ActiveIngredient*, that in *Schema.org* is just a property of *Drug* but, since it is crucial in the query construction, we choose to create it as a class, and for *DrugWithoutPrescription* that is not present in *Schema.org*. Other differences between our Owl and the one of *Schema.org* is at the data property level, where we decided to extend the model with new attributes to obtain more specialized classes.

This approach results in a better implementation of our ontology with *Schema.org*, as we can observe from Figure 5, where our custom entity are correctly linked to the top-level ontology.

3.4. Model visualization

Here we can appreciate a visualization of our final ER model (Figure 3). As we can see the main relationships are between *Drugstore,* and *Website* with *Drug*, in line with the main purpose of the final product: smartRemedy.

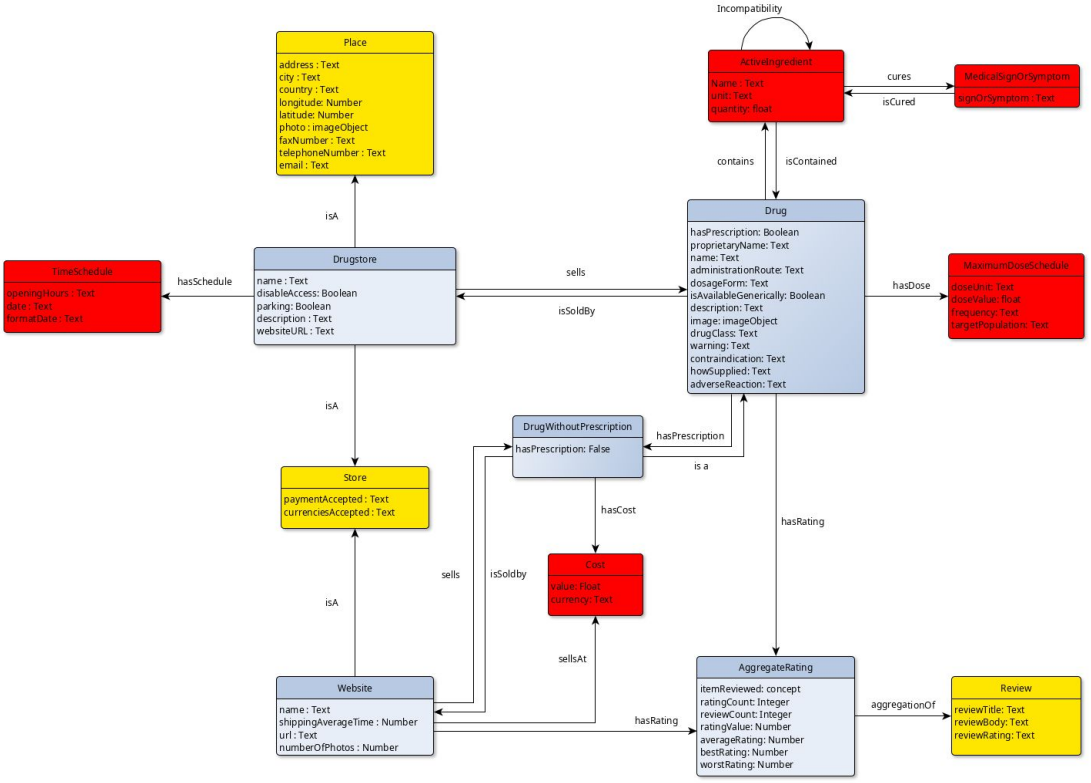


Figure 3. ER model created with yEd tool.

For build our ontology we use the tool Protégé as illustrated in section 3.2. Model formalization description. The resulting ontology is represented as follows (see Figure 4).

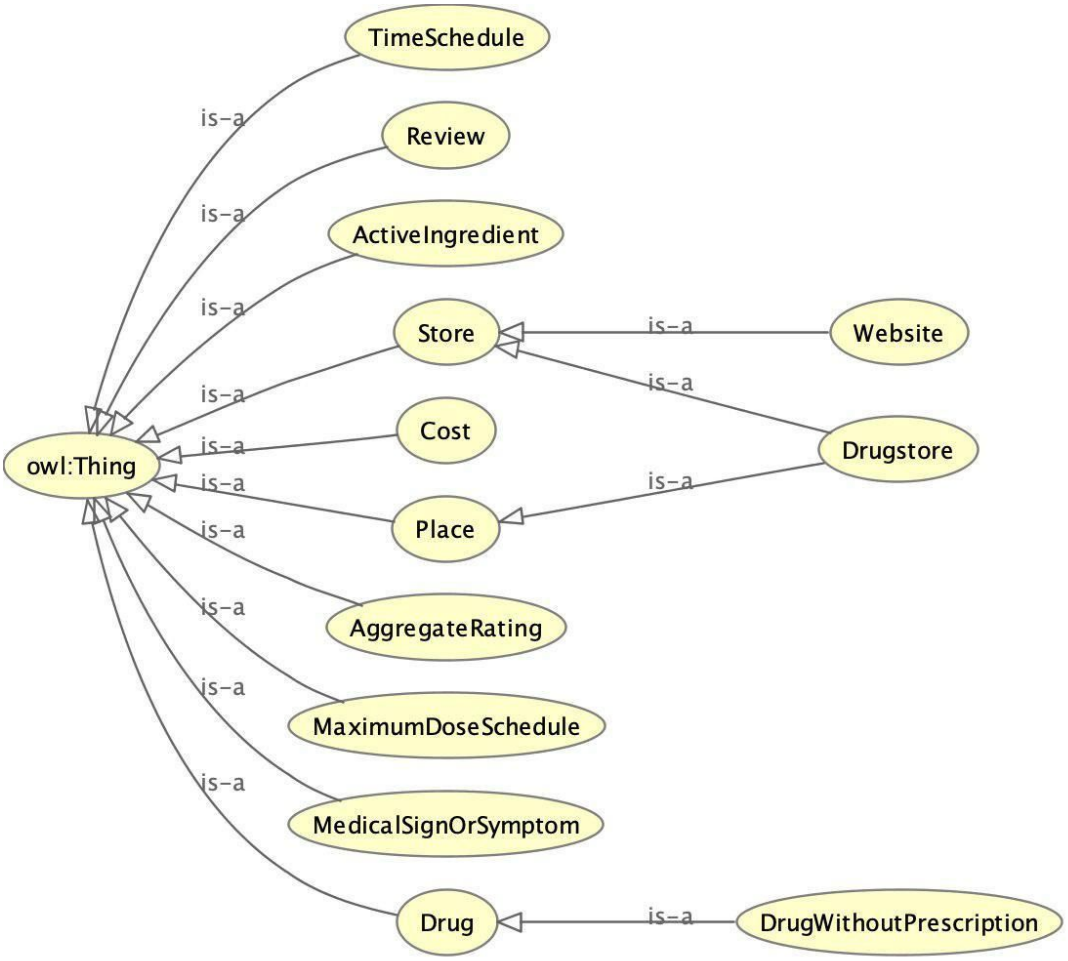


Figure 4. Model ontology obtained using Protégé tool.

Finally, we have the representation of the ontology model integrated with the top-level ontology *Schema.org*. We can observe how the complexity of the ontology increases including all related super-classes of *Schema.org* (see Figure 5).

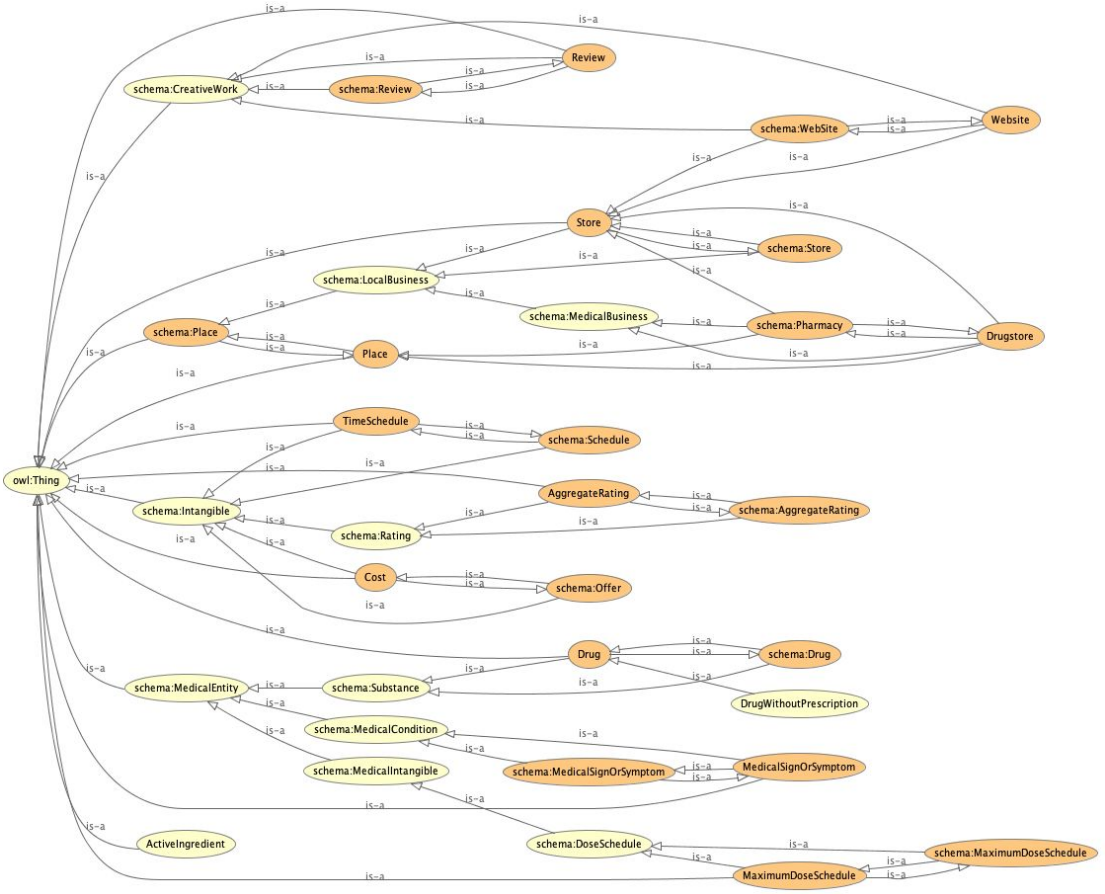


Figure 5. Final model ontology obtained using Protégé tool.

4. Data Integration

4.1. Select dataset(s)

For our needs, we've chosen to utilize the dataset from *DrugCentral.org* for the information about the drugs and their relation with the symptoms they cure. This dataset is a PostgreSQL Database containing 53 tables and a total of over 2.800.000 records.

This database contained only generic information about drugs, so to create a more useful application we've decided to integrate the data from famous American online shops that sell the majority of the drugs contained in the DrugCentral database. From our researches, websites that have the most drugs availability are *Walmart* and *Walgreens*. Since there was no direct access to a dataset of this data and available APIs where limited and had to be requested (a process that could take too much time for the scope of this project) we decided to scrape the data directly.

To obtain data about the local drugstores, since in Italy there's no a predefined database with the openings of the drugstores due to their continuous variations, we've decided to use farmaciediturno.org that is continuously updated giving us useful information about drugstores available in the territory and their opening hours. Moreover from the information like address and city, we can then easily obtain the latitude and the longitude (thanks to *OpenStreetMaps*) to allow us to add, in the end, the possibility to find the nearest drugstore thanks to their coordinate.

We also chose to add review data for the websites from *trustpilot.com* and reviews for the drugs from *webmd.com* and *everydayhealth.com*.

4.2. Data extraction

The database from *DrugCentral.org* is huge, and it contains a lot of chemical data that for our application are not necessary. After installing a local PostgreSQL service and loaded the database in our computers, we've started to search for all the tables and then the single attributes that we need for our project and start learning how the database is disposed. In particular, from that database, the most important information we've got is about: Drugs (with the related information like name, form, prescription

needed, etc), Symptoms, Package Leaflets, and all the tables that represent the relations between those elements.

For both *Walmart* and *Walgreens* data, we've chosen to write a scraper that, in both sites, research each product in the DrugCentral database (searching for its name), and give us all the information we need: price, name, rating, URL, photo, reviews.

In the case of *Walmart*, the scraper was written in Python, using BeautifulSoup with Selenium as the main scraping library. In the case of *Walgreens*, instead, we've written it in NodeJS using *Puppeteer*. Both use a headless Chrome instance to get the content page, and that was necessary since the pages are dynamically populated and with lower level scraping libraries (as BeautifulSoup alone for Python or fetch for NodeJS) we weren't able to get all the content.

For *farmaciediturno.org*, instead, we've written a scraper in Python, but thanks to the static nature of the pages in that case we've used BeatifulSoup alone. In the same script, when we obtain a new address, from the *OpenStreetMaps* API we collect the coordinates too. Since we have to scrape *farmaciediturno.org* every day to collect the updated opening hours for all the drugstores, we've set up a cache system that saves the coordinates of the address so it doesn't have to request every time the same information but instead only when needed, so basically when a new unknown address is found.

For gathering and preprocessing reviews data regarding websites we decided to scrape them from trustpilot.com, since it contains a huge number of reviews for almost every website available. Given the name of the website (for example, Walmart), it scrapes all the reviews of Walmart in the Trustpilot website, and the said data is then stored in a JSON file.

The libraries used are “pandas” for the data analysis, “json” for saving the information in a well defined structured JSON file, and as well for the other python scrapers “BeautifulSoup”, for the scraping from the website.

For the reviews of the drugs, instead, we scraped *webmd.com* and *everydayhealth.com* websites, again with python scrapers. For webmd and everydayhealth we’ve used “scrapy”, because of the dynamicity of the page. In all the cases, we’ve used the “json” library in order to save the processed results in a JSON file.

For the photos of drugstores and drugs, we created a python scraper, which given the query name, returns all generic information of the first two photos returned by google image search. This was done thanks to a library called “google_images_download”. The “json” library was used then to store the processed results in a JSON file.

4.3. Data preparation/Analyze the output data

In the DB of *DrugCentral.org*, we've started cleaning all the tables, that we don't need, and in the remaining tables we've removed the superfluous attributes. Since the database is extremely huge, we’ve started to filter out some of the data in order to obtain an outcoming RDF that is not extremely heavy. We’ve started filtering the symptoms, taking only 70 from all of them, and then filtering out the data consequently, so basically taking only the active principles that cures those specific symptoms, and then only the drugs that contain one or more of those active principles, and so on. In that way, we’ve reduced drastically the dimension of the db, in order to obtain a smaller and more manageable RDF. Obviously, in a real-world application, we should have used all the available data.

We've collected all the other data coming from our scrapers in JSON format, and since we've modelled the output of our scripts based on our needs, the preprocessing phase was not needed for those.

One problem we had to face was how to organize the data in order to represent the relations present. For “one to many” relations we used the flexibility of json files by including a list of all the destination objects into a field of the source one therefore completely describing the relation. For “many to many” relations we solved our issues thanks to the use of unique identifiers for objects. In this way we kept the objects separate and add a “relation object” composed by only the UIDs of the two entities to link.

4.4. Related ontology proposal

The main reference since the start of the project was *Schema.org*, since between all the upper-ontologies that we’ve checked is the one that contains the most of our classes and so is one of the most already compatible. Because of that, we’ve modelled our data respecting the structures already found in the *Schema.org* ontology. For our

needs, thanks to its broad scope, we think that *Schema.org* is the most appropriate ontology to be connected on ours. In facts, it already contains some of the medical classes that we need, such as *ActiveIngredient*, *Drug*, *MedicalSignOrSymptom* and *MaximumDoseSchedule*, but even the ones regarding the sellings as *Cost*, *Store*, *TimeSchedule* (for the openings of the store), *Website* and *Review*. And in all of them, the properties are similar to the ones that we need and so that we've included in our ontology.

SUMO contains some of the classes that we've included in our ontology too, but not as much as *Schema.org*. In fact, we have classes like *Drugstore*, *Medicine* and *BiologicallyActiveSubstance*, but it misses all the classes of the selling domain regarding for example reviews, costs, ratings, etc.

DBPedia, for its nature, contains classes like *Drug*, but most of its attributes are more chemical/scientifical related. Since the data is extracted from Wikipedia, is missing the selling domain too and so there are no reviews, costs and photos related classes.

4.5. Data integration with KARMA

After scraping and collecting all data of the reviews and drugs, we decided to integrate everything into a single JSON file called *alldata.json*. More precisely, this file is a single JSON object with the following keys: *farmacie*, *siti*, *medicine*, and *farmacieselling*.
The key *farmacie* contains the *Drusgtore* entities, and each *Drusgtore* contains multiple *TimeSchedule* entities. The key *siti* contains the *Website* entities and *AggregateRating* entities, which are related to the *Website* entities. Each *Website* contains the *DrugWithoutDescription* entities, and each entity *DrugWithoutDescription* contains *AggregatetRating* entities, which in turn contains the *Review* entities. The key *medicine* contains all the *Drug* entities. Each *Drug* entity contains *AggregateRating* entities, which in turn contains the Review entities. The key *farmacieselling* contains the *Drugstore* entities and the *Drug* entities, and we created this key to represent the relationship one-to-many between *Drugstore* and *Drug*. As we agreed with the professors, we assumed that each drugstore sells any type of drug.
One important note is that for the reviews, we had mainly three different sources of information: trustpilot.com, webmd.com, and everydayhealth.com. Luckily, the reviews of the websites were very similar; they had the same system rating (based on

stars), same range of evaluations (from 1 to 5), except with WebMD that has three different types of evaluation: one for effectiveness, one for satisfaction and the other one for ease of use. Due to the nature of the project, we decided to consider only the satisfaction value for reviews. The fact that they came from different sources imposed us to create a standard format for the reviews and the aggregate ratings collected, with respect to the *Review* and *AggregateRating* classes defined in the file .owl generated from the Protégé software.
In the first version of our *alldata.json* file, we did not specify any unique ID except for the ones defined for drugstore and in the DrugCentral dataset for the drug. For generating unique IDs for each review, aggregate rating, and opening hours, we used the uuid module of Python (the uuid3 method, to be precise), which allows us to generate a unique ID based on the MD5 hash of a namespace identifier and a string. We've made sure that the same object would generate the same ID if the uuid3 is applied multiple times on said object. After that, we used Karma to associate the column IDs with the URIs of the classes. Thanks to that, we could cross-reference each entity.
We modified the *alldata.json* file multiple times: the other sub-group, understanding that the same product could have different prices, created a new class called *Cost* containing the price and source link for the drugs without prescriptions, and they added a few properties in the *Drug* class in Protégé. So we had to create new unique IDs in the data for cross-referencing the entities of the new classes and associate the columns data to the new properties. Thanks to constant communication with them, we were also able to fix visible and hidden errors of the classes defined in the owl file. There are cases in which some classes are subclasses of others in Protégé (e.g., the *Drugstore* class is a subclass of *Store* and *Place*). In those situations, in Karma we defined the relationship between columns of data and properties of the subclasses directly.
Another critical problem we faced during the integration process was to express in Karma the one-to-many relationships. We managed it by storing multiple entities of a relationship into a JSON array with a key-value association. For example, let us consider the relationship *hasOpening* between the *Drugstore* and *TimeSchedule* classes. Besides the data that is associated with the properties of the class Drugstore, there is also an *openingHours* key in which we stored a JSON array of objects, and each JSON object describes the *TimeSchedule* entity. We used this approach for almost all the relationships between the classes.
The entity *Website* has multiple *AggregateRating* entities, and each *AggregateRating* has multiple *Review* entities. By using the URIs correctly, we represented the inverse: the same entity, which is represented as a JSON object, is stored in multiple different

JSON arrays, and this creates a one-to-many relationship to the objects that contain it. This was useful in cases such as the *isCured* relationship between the *MedicalSignOrSymptom* and *ActiveIngredient* classes: we could express the fact that a *MedicalSignOrSymptom* entity can be cured by multiple *ActiveIngredient* entities.
We had problems with *isIncompatibleWith* because it was a relationship one-to-many between *ActiveIngredient* entities only. In Karma, we define a new *ActiveIngredient* class, which Karma labels as *ActiveIngredient1*. We associate columns of data to the properties of the *ActiveIngredient* class, but in the data therei is also defined an array that contains the IDs of the other *ActiveIngredient* entities that are incompatible with the *ActiveIngredient* entity that contains them. We then create a new *ActiveIngredient* class, which Karma labels as *ActiveIngredient2*. We associate the IDs in the array to the URI of the *ActiveIngredients2* class. Then, we make the isIncompatibleWith link that starts from *ActiveIngredient1* class to *ActiveIngredient2* class. The process of associating the data to the properties of the classes was quickly done, and we explicitly define the types of literals, in respect to the XML Schema (e.g., when associating integer data to a property of a class, we define the type of literal of the link as xsd:integer).
When we started to approach to the software Karma, one problem that immediately arose was that despite the fact that the other sub-group labeled the classes in Protégé (rdfs:label) the linked data file generated from Karma did not contain any labels. Thus we solved this by manually assign for each class in Karma the rdfs:label property. We were not able to understand why this was happening and to figure out an efficient solution; it was probably an error of Protégé or the generated owl file. We find the solution thanks to an inefficient trial and error method applied both on the .owl file and the RDF file. Moreover, the fact of using basic functions offered by Karma, such as *PyTransform*, which seems to be useful, can create abnormal and unexpected results. *PyTransfom* allows us to execute small python scripts, useful for creating new columns with a generated value inside. We decided that for each entity, a better way to identify its URIs was concatenating their entity name with their unique ID (for example, drugstore/{id}, review/{id}, etc.). We used *PyTransfrom* applied directly to the ID columns data in order to create the new URIs of the classes, given their IDs, but the problem is that its behavior did not respect the history of commands in Karma. We got some duplicated IDs on some entities in the linked data output file. Again, how could we find what we were doing wrong? After several attempts, we tried with a new approach. Instead of applying the scripts directly to the column containing the IDs, we created new columns with the modified URIs that we wanted. With this method, we finally got the expected URIs. After all these numerous changings, the final RDF file generated from Karma was what we expect to be.

4.6. Knowledge Graph description

From the outcoming Knowledge Graph outputted from Karma, we've started to explore all the information contained in it with *GraphDB* in order to check if all our queries were possible. In fact, it was a success, since, thanks to the querying of the knowledge graph in *SparQL*, we were able to obtain all the information we needed.

The graphical representation of the Karma Model, generated directly by Karma as a .dot file which was then rendered using GraphViz with the command "`dot -Tpng model.json-model.dot -o model.png -Grankdir=LR -Goverlap=false -Ksfdp`", is represented in Figure 6. After importing the knowledge graph in GraphDB, we were able to get a representation of some of the instances of our graph, as illustrated in Figure 7.

Those are the main queries in *SparQL* language that we've used as an example to demonstrate the functionalities of the knowledge graph:

- *Query to obtain all the drugs that cures the symptom "cough"*

```
PREFIX : <http://www.semanticweb.org/kdi#>
select * where {
      ?syn a :MedicalSignOrSymptom .
      ?syn :signOrSymptom ?name .
      ?syn :isCuredBy ?ai .
      ?ai :nameAI ?ai_name .
      ?ai :isContained ?drug .
      ?drug :nameDrug ?d_name .
      ?drug :administrationRoute ?route .
      FILTER (str(?name) = "cough")
}
```

- *Query to obtain all the drugs that cures the symptom "cough" but the administration route is not "ORAL"*

```
PREFIX : <http://www.semanticweb.org/kdi#>
select * where {
      ?syn a :MedicalSignOrSymptom .
      ?syn :signOrSymptom ?name .
      ?syn :isCuredBy ?ai .
      ?ai :nameAI ?ai_name .
      ?ai :isContained ?drug .
      ?drug :nameDrug ?d_name .
```

```
        ?drug :administrationRoute ?route .
        FILTER (str(?name) = "cough")
        FILTER (str(?route) != "ORAL")
    }
```

- *Detailed description of the drug "Tussin"*

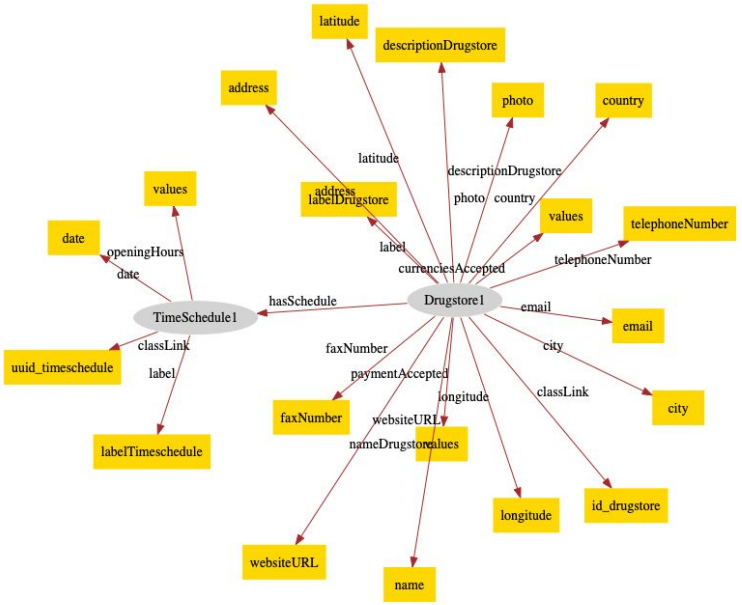
```
    PREFIX : <http://www.semanticweb.org/kdi#>
    select * where {
        ?drug a :Drug .
        ?drug :nameDrug ?name ;
        :administrationRoute ?route ;
        :descriptionDrug ?description;
        :hasPrescription ?prescription;
        :isAvailableGenerically ?available_gen;
        :proprietaryName ?pr_name;
        :dosageForm ?dosage;
        FILTER(str(?name) = "Tussin")
    }
```

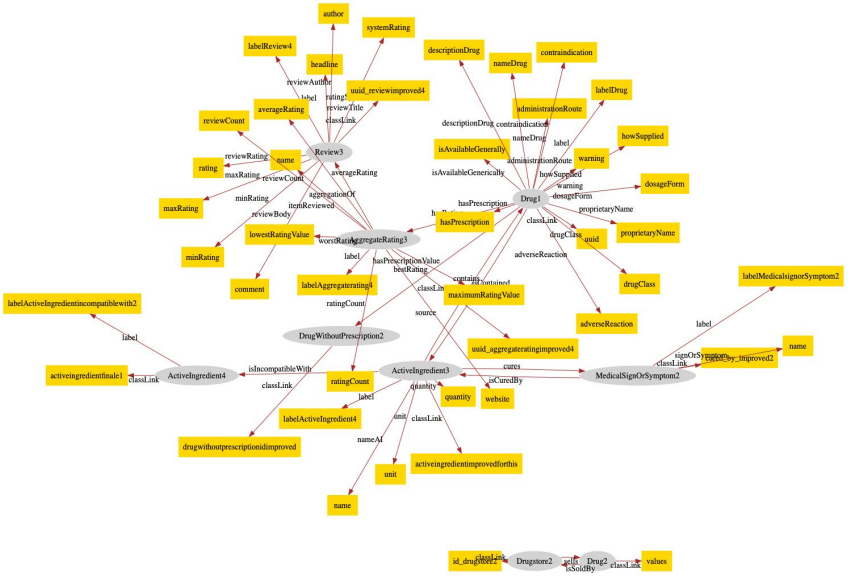
- *List of local drugstores*

```
    PREFIX : <http://www.semanticweb.org/kdi#>
    select * where {
        ?phar a :Drugstore;:nameDrugstore ?name ;
        :address ?addr ;
        :city ?city ;
        :country ?country ;
        :latitude ?lat ;
        :longitude ?lon ;
        :hasSchedule ?sche ;
    }
```

- *Price and websites of the drug called "Tussin"*

```
    PREFIX : <http://www.semanticweb.org/kdi#>
    select * where {
        ?p a :DrugWithoutPrescription ;
        :nameDrug ?name ;
        :hasCost ?cost ;
        :isSoldBy ?web .
        ?cost :value ?price .
        ?web :nameWebsite ?w_name
        FILTER regex(str(?name),  ".*Tussin.*")
    }
```


latitude
descriptionDrugstore
address
photo
country
values
telephoneNumber
date
openingHours
labelDrugstore
label
currenciesAccepted
telephoneNumber
TimeSchedule1
hasSchedule
Drugstore1
email
email
uuid_timeschedule
classLink
label
faxNumber
paymentAccepted
city
city
longitude
classLink
faxNumber
websiteURL
nameDrugstorealues
labelTimeschedule
longitude
id_drugstore
websiteURL
name


author
systemRating
labelReview4
headline
uuid_reviewimproved4
averageRating
reviewCount
Review3
name
rating
maxRating
minRating
lowestRatingValue
comment
labelAggregaterating4
AggregateRating3
reviewBody
itemReviewed
aggregationOf
DrugWithoutPrescription2
labelActiveIngredientincompatiblewith2
activeingredientfinale1
ActiveIngredient4
isIncompatibleWith
classLink
ratingCount
labelActiveIngredient4
drugwithoutprescriptionidimproved
nameAI
unit
name
ActiveIngredient3
quantity
website
activeingredientimprovedforthis
descriptionDrug
nameDrug
contraindication
labelDrug
administrationRoute
howSupplied
warning
isAvailableGenerally
isAvailableGenerically
dosageForm
Drug1
hasPrescription
proprietaryName
drugClass
uuid
adverseReaction
maximumRatingValue
uuid_aggregateratingimproved4
cures
isCuredBy
MedicalSignOrSymptom2
labelMedicalsignorSymptom2
signOrSymptom
labelSignorSymptom_improved2
name
id_drugstore2
Drugstore2
sells
isSoldBy
Drug2
values

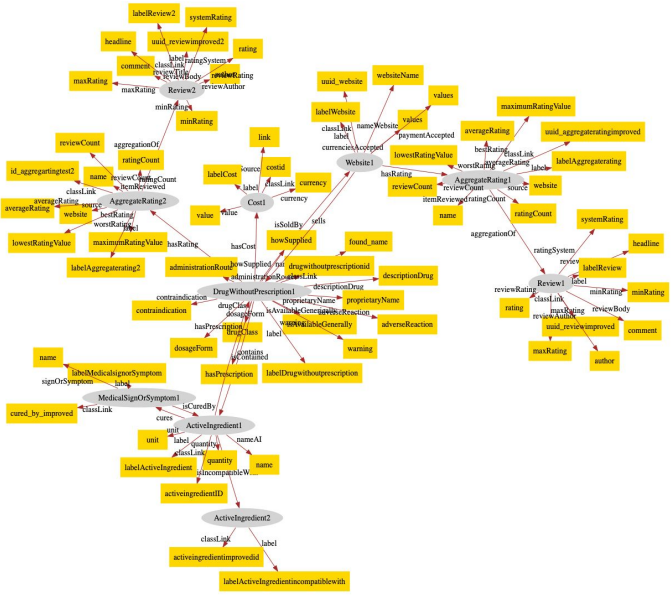


Figure 6. Those three graphs are the Karma Model exported directly from Karma as a dot file and then rendered using Graphviz. The image is splitted in three different part only for impagination purposes.

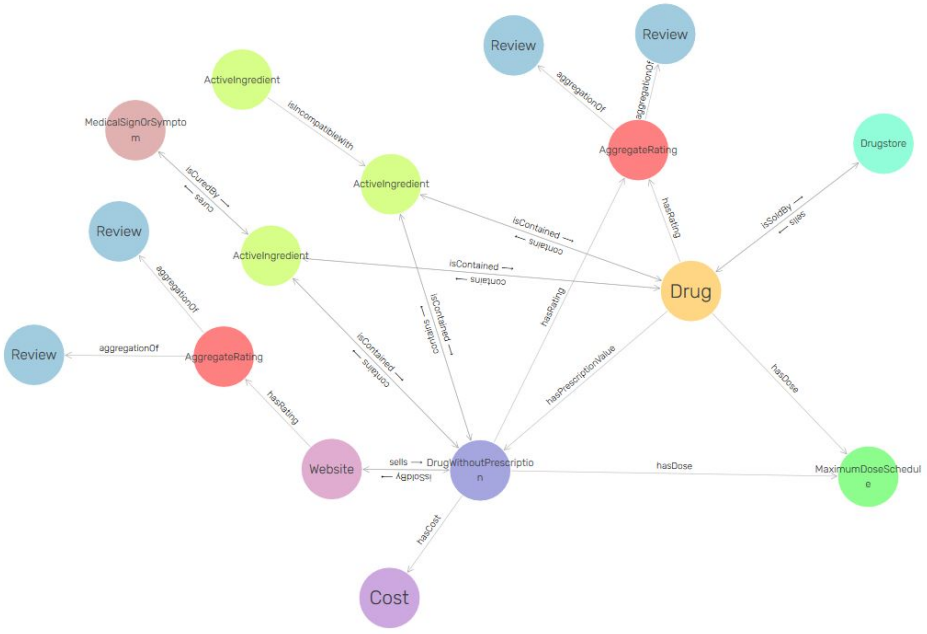


Figure 7. Representation of some instances of the Knowledge Graph made with GraphDB

5. Final considerations and open issues

In our data, we've filtered out a lot of data from the database of DrugCentral in order to obtain a more manageable RDF file. In a real-world application, we should have used all of the information available. Moreover, a better approach that allows gaining even more precise relationships between the drugs and the symptoms could be to scrape from the various web stores all the information about the drugs (as we made) and getting the symptoms that it cures directly from there too. Since in all the web stores generally, we have the category of the drug that represents the symptom that it cures, it's easy to scrape the category and from it create the association (even considering synonyms of the symptom and so on). In that way, the result is way better appropriate, since the drug is effectively related to the specified symptom. Merging the database from DrugCentral and the webstores, instead, produced some wrong relationships, because if the product was not sold by a specific website, in the scraping process of that website, when the research of the product name was done, the website gave us some different products. Even comparing the differences in the name, is not so easy to understand if the match is a good match or not. We were aware of that, but we wanted to use in any case the database for learning purposes in order to understand how to clean and integrate multiple sources together. For the future, as made for Walgreens and Walmart, the idea is to integrate even more websites, and if we consider the approach of taking the symptoms directly from the website itself, that allows us to consider even Italian websites and Italian drugs. Ideally, collected data can be structured in order to provide further informations regarding the usage, in order to fully satisfy our model. Examples of this can be retrieve specific information from a drug without passing through the entire package leaflet, retrieve information regarding the drugstore accesses or services, etc. Furthermore, in our knowledge graph we're assuming that every local drugstore sells every drug: a cool approach in a real-world environment could be to create some connection to the local pharmacy in order to obtain for each the full list of drugs that they currently have in-store, so when someone search for a specific drug can see what is the nearest local pharmacy that actually has that specific drug available.