Web Sémantique

Projet MovieRetriever

2017

CONTENU

Ce rapport est une présentation de notre projet et la liste des spécifications pour celui-ci.

Par : Rémy AUDA et DRAVET Jean-Baptiste

Université Nice Sophia Antipolis

Table des matières

Présentation projet 3
Description 3
Explication 3
Consignes 3
Ontologie 3
Les classes 3
Les propriétés d'objets 4
Le projet 4
Récupérateur des données 4
Architecture 4
Algorithmes 5
Chargement des données 5
Problèmes 5
Initialisation 6
Remplir l'ontologie 8
Création 9
Conclusion 10
« Akinator » 10

Présentation projet

Le but de notre projet est de réaliser un logiciel qui permet de retrouver un film en répondant par oui ou par non à des questions qui seront posées.

Description

Le principe de notre logiciel est, via une interface graphique java, de poser des questions à l'utilisateur à propos d'un film afin de retrouver le titre du dit film. Pour cela, nous allons créer notre propre ontologie simplifiée.

Il s'agit d'un logiciel dans l'esprit de l'Akinator, à la différence qu'avec un Akinator, nous sommes en connaissance du film et ce serait à l'Akinator de le deviner alors qu'ici c'est l'inverse : la personne n'est pas en connaissance du film et notre logiciel essaye d'aider la personne à trouver le titre du film.

Explication

Nous avons décidé de ne pas récupérer une ontologie existante car elles nous ont semblé trop complexes avec beaucoup trop d'informations. En effet lorsque l'on recherche un film, on a souvent peu d'informations à son sujet. Poser des questions à propos de la musique ou des costumes nous a donc paru inutile. De plus, les grosses ontologies ont un système complexe d'ID (identifiant) qui lie les différentes classes entre elles, ce qui nous semble trop complexe à implémenter dans le temps imparti.

Consignes

Notre projet est lié aux consignes suivantes :

- RDF data
- RDFS / OWL / SKOS vocabularies, rules
- SPARQL queries (and show the inferences made, taking into account the semantics of the vocabulary)

Nous avons décidé de réaliser notre ontologie en OWL avec l'aide du logiciel Protege.

Ontologie

Voici une représentation de notre ontologie minimaliste. Cette ontologie est utilisée uniquement dans le but de notre logiciel. Elle se justifie par l'utilisation que nous en faisons.

Les classes

Voici la liste des différentes classes que nous avons créées :

- Actor : classe pour représenter les acteurs d'un film. Chaque acteur sera uniquement décrit par son nom, les films dans lesquels il a joué et le personnage qu'il a joué dans chaque film.
- Award : représente les récompenses pour les films.
- Characters : représente les personnages des films. Ils sont joués par des acteurs.
- Country : représente le pays d'origine d'un film.
- Movie : classe qui représente les films. En relation avec toutes les autres classes de l'ontologie.

- Producer : représente les producteurs pour chaque film. Ils font partie de la Team.
- Scriptwriter : représente les scénaristes pour chaque film. Ils font partie de la Team.
- Team : représente l'équipe humaine d'un film.
- Type : représente le type de film (le genre auquel il appartient).

Les propriétés d'objets

Voici la liste des différentes propriétés que nous avons créées :

- Characters : relation en un personnage et un film. Définit qu'un personnage appartient bien à un film.
- OriginalCountry : relation entre un film et son pays d'origine.
- hasActor : relation entre un film et un de ses acteurs. Inverse à la propriété hasPlayIn.
- hasPlayIn : relation entre un acteur et un des films dans lequel il a joué.
- hasScriptWriter : relation entre un film et son scénariste. Inverse de la propriété wasScriptWriterIn.
- wasScriptWriterin : relation entre un scénariste et un de ses films.
- isTypeOf : relation entre un film et un de ses types (action, aventures, …)
- wasPlayedBy : relation entre un personnage et l'acteur qui le joue.
- wasProducerIn : relation entre un producteur et un de ses films. Inverse de la propriété wasProductBy.
- wasProductBy : relation entre un film et son producteur.
- wasRewardWith : relation entre un film et une de ses récompenses.

Nous avons décidé d'ajouter un poids sur chaque propriété et chaque valeur. Ces poids vont nous servir pour réaliser un algorithme de pseudo-génération des questions.

Cette partie sera développée ultérieurement dans ce rapport.

Le projet

Nous avons décidé de réaliser le projet en Java avec Jena pour la réalisation des requêtes SPARQL. Suite à la discussion avec notre intervenant, nous allons devoir diviser notre projet en deux projets bien distincts.

Récupérateur des données

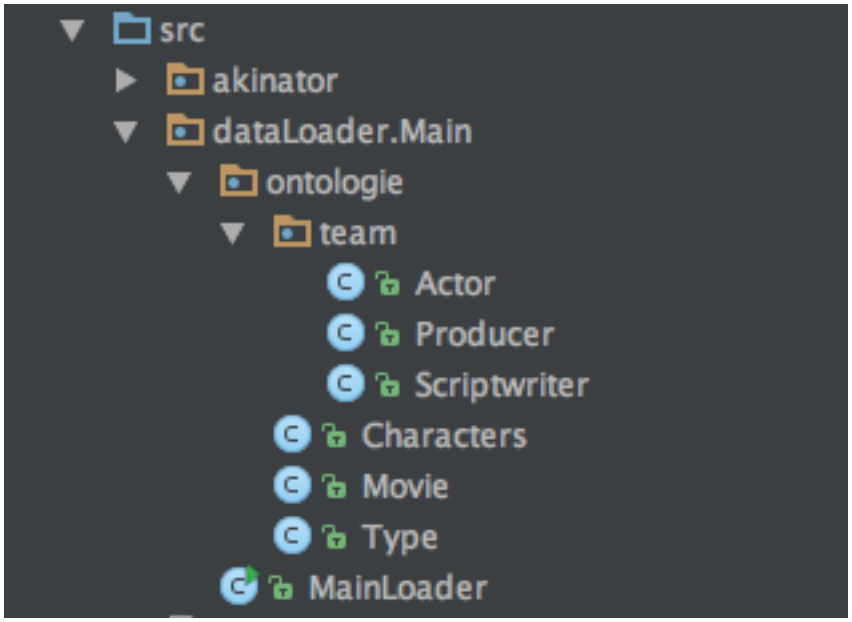
Dans un premier temps, nous allons devoir créer un projet de récupération de données qui va, grâce à un dump d'un dataset, récupérer les données des bases de connaissances déjà existantes et remplir notre propre base de données. Pour se faire, il va falloir réaliser un mapping entre l'ontologie de la base de connaissance et la nôtre. Grâce au mapping et à des requêtes SPARQL, nous allons pouvoir remplir une base de données configurée pour notre propre ontologie.

Architecture

Pour réaliser le loader, nous avons réfléchie à une architecture pour répondre à la question de représentation et du mapping. Plus clairement, comment allons-nous faire la

transition entre les données sous forme de tuple dans un dataset à des instances de données dans notre ontologie.

Voici un screenshot de l'architecture de notre sous-projet « Loader » :



Chaque classe portant un nom correspondant à une classe de notre ontologie va nous permettre de faire le traitement de la donnée en liaison direct avec la classe. Par exemple, « Actor » s'occupera de représenter les acteurs, d'ajouter les acteurs dans l'objet « Movie » et de rajouter les instances d'acteurs dans notre ontologie.

Donc nous allons représenter chaque instance par un objet (une instance de donnée pour un acteur va être utilisée pour créer l'objet acteur). Ensuite, l'objet Movie sera composé de plusieurs instances de différents objets (plusieurs types, plusieurs acteurs). Enfin, nous allons dans le « MainLoader » charger notre ontologie et la remplir avec tous les objets que nous avons créés.

Algorithmes

Chargement des données

Nous avons tout d'abord voulu récupéré les informations depuis un endpoint sparql de linked database Movie. Nous avons choisi cet endpoint car son ontologie était la plus proche de la nôtre.

Problèmes

Après plusieurs essaies de récupération, nous sommes tombés sur un problème majeur : nos requêtes successives se faisaient déconnecté car le site pensait qu'on essayait de le DDOS (flux trop important de requêtes).

Après avoir changé notre façon de faire les requêtes, nous avons réussi à faire passer plus de requêtes mais nous avons aussi réussi à faire tomber le site.

Nous avons alors demandé l'aide de notre intervenant. Il nous a conseillé d'utiliser un dump de linked data movie, ce que nous avons fait. Nous avons enfin pu récupérer toute les données que nous voulions.

Nous avons encore eu quelques problèmes d'URI mal formé dans le dump, que nous avons corrigé à la main.

Initialisation

Pour récupérer les données, nous avons d'abords dû charger les données. Pour cela nous avons utilisé la classe Model.

```
// initialisation /////////////////
Model model;
final String inputFileName  = "File/RDFS_file/linkedmdb-latest-dump.nt";

// créer un modèle vide
model = ModelFactory.createDefaultModel();

// utiliser le FileManager pour trouver le fichier d'entrée
InputStream in = FileManager.get().open( inputFileName );

// lire le fichier owl
model.read(in,null, "N-TRIPLE");
```

Après avoir chargé le dataset, nous chargeons notre ontologie dans un model diffèrent :

```
//load of our ontologie
Model m = ModelFactory.createDefaultModel();
final String inputFile  = "/Users/titanium/Desktop/MoviesInformationRetriever/File/RDFS_file/OntologieMovie.owl";

InputStream test = FileManager.get().open(inputFile);
m.read(test, "RDF/XML");
System.out.println("fini load ontologie");
```

Maintenant, nous allons créer une arraylist de chaque classe pour récupérer les instances qui sont contenu dans le dataset et les transformer en objet :

```
System.out.println("Create Character list");
characterses = Characters.constructListOfCharacter(query,queryexec,resultSet,model);

System.out.println("Create producer list");
producers = Producer.constructListOfProducer(query,queryexec,resultSet,model);

System.out.println("Create type list");
types = Type.constructListOfType(query,queryexec,resultSet,model);

System.out.println("Create writer list");
scriptwriters = Scriptwriter.constructListOfWriter(query,queryexec,resultSet,model);

System.out.println("Create actor list");
actors = Actor.constructListOfActor(query,queryexec,resultSet,model);

System.out.println("Create movie list");
movies =  loadMovieFromBigDataset(model);

System.out.println("add all  to ontologie");
Movie.fillOntologie(movies,types,actors,characterses,producers,scriptwriters);

System.out.println("ends");
```

Chaque classe possède donc une méthode qui permet de récupérer les informations que nous voulons grâce une requête Sparql.

Voici un exemple de requête pour Acteur, qui ressemble beaucoup aux autres requêtes pour chaque classe :

```
public static String requestforinstance = "prefix mo:  <http://data.linkedmdb.org/resource/movie/>" +
        "prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>" +
        "SELECT DISTINCT  ?name WHERE" +
        " { ?uri  mo:producer ?uriProducer. " +
        "?uri a mo:film. "+
        "?uri mo:initial_release_date ?date."+
        "?uri mo:runtime ?duration."+
        "?uri mo:country ?countrytemp."+
        "?countrytemp rdfs:label ?country." +
        "?uri mo:genre ?genre." +
        "?uri mo:writer ?uriWriter." +
        "?uri mo:actor ?uriactor." +
        "?uriactor rdfs:label ?name." +
        " } ";
```

?uri représente l'uri d'un film. Pourquoi une tel requête pour les acteurs ? Car nous souhaitons obtenir uniquement les acteurs des films que nous aurons déjà présélectionné.

Les films présélectionnés devront avoir au minimum un pays d'origine, une date, une durée, et un titre. Tout film n'ayant pas au minimum ces 4 critères ne sera pas récupéré.

Donc pour avoir la liste d'acteur uniquement lié à ces films, nous allons utiliser les paramètres de la requête pour les films comme « filtre » (sans être filter de sparql) et utiliser ceux-ci pour filtrer les acteurs que nous voulons récupérer.

Ensuite nous allons construire une liste d'acteur avec les résultats de la requête :

```
public static ArrayList<Actor> constructListOfActor(Query q, QueryExecution qr, ResultSet r, Model m) {
    String finalRequest = requestforinstance;
    ArrayList<Actor> actors = new ArrayList<>();
    q = create(finalRequest);
    qr = QueryExecutionFactory.create(q, m);
    r = qr.execSelect();
    while (r.hasNext()) {
        QuerySolution binding = r.nextSolution();
        Literal name = binding.getLiteral("name");
        actors.add(new Actor(name.getString().replace("(Actor)","")));
    }
    return actors;
}
```

Nous pouvons voir dans le code ci-dessous un code exploitant la librairie Jena permettant de manipuler le web sémantique.

Pour Movie, cela sera un peu différent. Nous allons d'abords récupérer chaque film et ensuite construire les différentes listes (acteurs, producteurs, scénaristes et types) liées au film en question :

```java
/**
 * load file from dump and return an arrays of movies
 * @param model
 * @return
 */
public static ArrayList<Movie> loadMovieFromBigDataset(Model model){
    Query query = null;
    QueryExecution queryexec = null;
    ResultSet resultSet = null;
    ArrayList<Movie> movies;
    movies = Movie.constructListOfMovie(query,queryexec,resultSet , model);
    movies = Movie.refactorListofMovie(movies);
    for (Movie movie: movies
            ) {
        movie.addListOfType(Type.constructListOfTypeForMovie(movie.getUri(),query,queryexec,resultSet,model));
        movie.addListOfScriptwriter(Scriptwriter.constructListOfscriptwritersForMovie(movie.getUri(),query,queryexec,resultSet,
        movie.addListOfProducer(Producer.constructListOfProducerForMovie(movie.getUri(),query,queryexec,resultSet,model));
        movie.addListOfActors(Actor.constructListOfActorForMovie(movie.getUri(),query,queryexec,resultSet,model));
        movie.addListOfCharacters(Characters.constructListOfCharactersForMovie(movie.getUri(),query,queryexec,resultSet,model))
       // System.out.println(movie.toString());
    }
    return  movies;
}
```

Les méthodes « addList. » permettent de récupérer les informations lié à l'Uri du film. Voilà par exemple la requête pour récupérer les types :

```java
//permet de récupérer les types pour un film donné
public static String requestpart1 = "prefix mo:  <http://data.linkedmdb.org/resource/movie/>" +
        "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>" +
        "SELECT DISTINCT  ?name WHERE" +
        " { <";
public static String requestpart2 =
        "> mo:genre ?urigenre." +
                "?urigenre rdfs:label ?name." +
                " }  ";
```

Remplir l'ontologie

Une fois que nous avons chargé nos fichiers et créé nos objets, nous allons créer les ressources dans notre ontologie. Pour se faire, pour chaque liste d'objet, pour chaque objet, nous allons créer une ressource et les propriétés qui vont avec et écrire dans notre fichier ontologie.

```
    for (Type type:types
            ) {
        type.addTypeToOntologie(m);
    }

    for (Actor actor:actors
            ) {
        actor.addActorToOntologie(m);
    }

    for (Characters characters1:characters
            ) {
        characters1.addCharacterToOntologie(m);
    }

    for (Producer producer:producers
            ) {
        producer.addProducerToOntologie(m);
    }

    for (Scriptwriter scriptwriter:writers
            ) {
        scriptwriter.addWriterToOntologie(m);
    }

    for (Movie movie:movies
```

Voici un exemple d'ajout de ressource pour un objet type :

```
// ajoute la liste d'instance de type dans l'ontologie
public void addTypeToOntologie(Model m) {
    String prefixemo = "http://www.semanticweb.org/titanium/ontologies/2017/0/untitled-ontology-11#";
    String prefixerdfs = "http://www.w3.org/2000/01/rdf-schema#";
    String prefixerdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";

    try {
        Resource resourceType = m.createResource(prefixemo + this.name);
        //type
        Property type = m.createProperty(prefixerdf + "type");
        resourceType.addProperty(type, prefixemo + "Type");
        //add title
        Property label = m.createProperty(prefixerdfs + "label");
        resourceType.addProperty(label, this.name);

    } catch (Exception e) {

    }
```

Création

Une fois toute les ressources créées, nous allons créer le fichier ontologie en utilisant notre model qui contient toutes les ressources.

```
try {
    m.write(new FileOutputStream("/Users/titanium/Desktop/MoviesInformationRetriever/File/RDFS_file/OntologieMovie1.owl"),
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Conclusion

Maintenant que nous avons notre ontologie avec une certaine quantité de donnée, nous allons l'utiliser pour notre deuxième partie : l'Akinator.

Cette partie de récupération de données qui nous venons de voir est très importante car elle illustre magnifiquement certains des avantages du web sémantique par rapport à d'autres technologies.

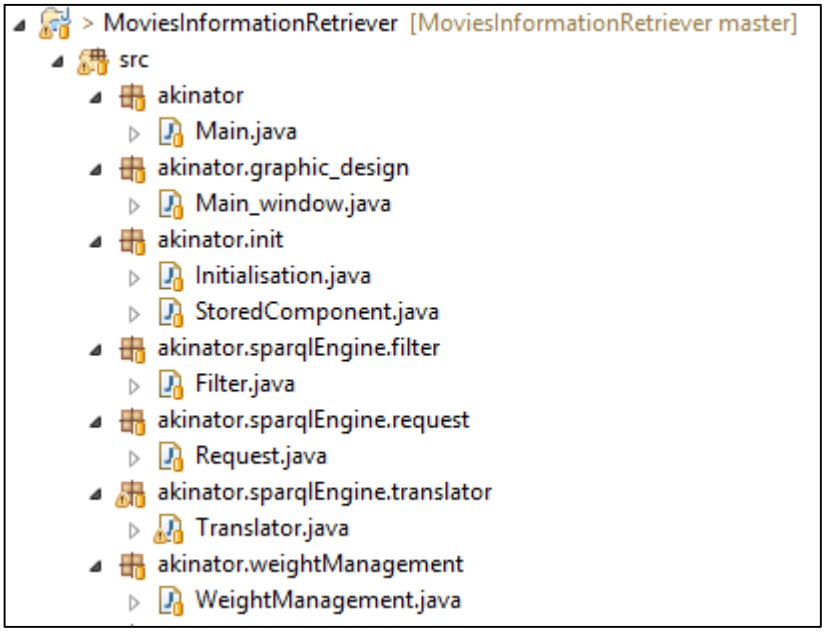
En effet, elle illustre le fait que nous pouvons charger des données directement dans le web. Pour le cas de notre application par exemple, à chaque fois que des films seront rajoutés sur le web, ces mêmes films seront donc ajoutés à notre application car nous chargeons les données directement depuis le web. Une application utilisant des bases de données par exemple ne pourrait bénéficier de ces avantages car à chaque fois qu'un nouveau film sortirait sur le web, il faudrait le rajouter à la main dans la base de données pour que l'application puisse prendre en compte ce nouveau film.

« Akinator »

Une fois les données récupérées, nous avons lancé le second projet. Il s'agit d'un « Akinator » qui va poser des questions à l'utilisateur en utilisant notre ontologie et nos données. Pour cela, nous avons introduit un système de poids. Chaque valeur et chaque propriété ont un poids. Ce poids va nous permettre de définir l'ordre des questions et la sélection des valeurs. Les propriétés qui auront le plus gros poids seront les premières utilisées pour la réalisation des questions. Une fois utilisé, nous allons mettre à jour le poids de la donnée avec des requêtes SPARQL de type DELETE/INSERT pour indiquer à notre algorithme que nous avons déjà utilisé cette propriété.

Architecture

Voici l'architecture pour le projet Akinator :



Problèmes

Aenean et interdum urna. Vestibulum eros orci, hendrerit commodo quam eget, faucibus elementum diam. Sed ut ullamcorper massa. Maecenas placerat pulvinar dolor et iaculis. Vivamus ante nisi, vehicula quis pulvinar quis, laoreet id enim. Sed maximus, enim et maximus lobortis, lectus dui sollicitudin nisi, id aliquet massa dolor sit amet sem. Vestibulum neque mauris, finibus quis libero sed, pulvinar egestas lacus. Vivamus massa nunc, aliquet et dui vitae, luctus tristique nisi. Pellentesque non dui ante. Quisque felis tellus, volutpat quis luctus sit amet, eleifend eget sem. Aliquam orci turpis, commodo vel nisi quis, dapibus pretium turpis. Mauris vitae laoreet mi.

Algorithmes

Idée principale

Pour réaliser l'Akinator, nous avons créé notre propre algorithme :

- On commence par récupérer la propriété avec le plus gros poids et la valeur qui a le plus gros poids pour la propriété en question.
- On récupère le label de chacun, on pose la question en utilisant les labels.
- En fonction de la réponse, on diminue le poids de la propriété et on met à 0 le poids de la valeur.
- On créer une requête pour récupérer notre film en fonction de la réponse à la question.
- Si on obtient un seul résultat, on l'affiche.
- Si on obtient plusieurs résultats, on repart à l'étape 1.

Ce système impose de mettre un poids sur chaque propriété et chaque valeur

Système de poids

Le système de poids est basique pour l'instant (on choisit un peu arbitrairement le poids à attribuer sur chaque valeur, avec un générateur de nombre aléatoire) mais dans une prochaine version, on peut imaginer que ce poids serait calculé en fonction de la fréquence de référence pour instance : par exemple, plus un acteur a joué de film, plus son poids sera conséquent.

Au fur et à mesure de notre avancement, nous avons réalisé l'erreur que nous avions faite en ne créant pas de classe « poids » dans notre ontologie. En effet, nous avons du coup mis le score du poids sur la propriété rdfs:seeAlso pour les valeurs et rdfs:IsDefiniedBy sur les propriétés. Mais nous avons alors rencontré un problème : les Data Property n'ont pas de rdfs:IsDefinedBy. Nous avons donc dû parer à ce problème en modifiant la donnée en elle-même : si la duration est égale à 1h30, alors nous modifions la donnée pour que la valeur soit « 1h30 OK ».

Ensuite, lors de la recherche par poids, nous allons faire ça dans l'ordre d'arriver normal, puis pour la décrémentation du poids, nous allons passer de « 1h30 OK » à « 1h30 NONE » pour ne plus le récupérer lorsque nous ferons une recherche pour la duration.

Vous l'aurez compris, la recherche sera différente si c'est une data Property ou une propriété de donnée. La query sera donc différente.

[Mettre Querry normal ici]

[Explication]

[Mettre Query OK ici]

[Explication]

.