Web data mining & semantics Final Project Report

By:

Manisha Priyadarshini Rawla

Shubham Saini

ESILV, DIA-2

CONTENT

Introduction

Data mining, commonly referred to as knowledge discovery from databases (KDD), is the process of nontrivial information extraction from data that is implicit, unknown, and potentially useful. Throughout the past few decades, developments in data mining techniques lead to many astonishing innovations in data analytics and big data. In order to evaluate massive data sets, data mining also combines methods from many different fields, including statistics, AI, machine learning, database systems, and many more. Data mining tasks that systematically include domain knowledge, particularly formal semantics, into the process are referred to as semantic data mining.

ROLE OF ONTOLOGIES IN SEMANTIC WEB

Ontologies are used in semantic data mining from different perspectives and in different ways depending on the systems and applications. There is no singular answer to the question of why ontologies are valuable in supporting data mining. We highlight the three goals that ontologies have been introduced to semantic data mining by evaluating the prior ontology-based methodologies. • To reduce the semantic disparity that appears between data, applications, data mining algorithms, and data mining outcomes. • To give a priori knowledge to data mining algorithms that either directs the mining process or reduces/constrict the search space. • To offer a formal method for expressing the entire data mining process, from data cleansing to mining results.

METHODOLOGY

In this report, we use the open movie dataset mentioned in DVO.

The project is done in 4 parts.
- Modeling the Ontology
- Populating the Ontology
- Querying the Ontology
- Manipulating the Ontology using Jena

We will Follow the steps mentioned on DVO to obtain the prescribed result.
Before moving forward the summary of the project is mentioned below.

Main objectives

- Design a movie application that follows the Linked Data principles: the application should be represented in a standard vocabulary that any application can process. As a starting point, movie instances should be described as instances of schema:Movie.
  Schema.org for Developers
  Schema.org en schemaorg.owl
- Develop an application to create, query, and validate calendar events.

Pedagogical objectives

- Do a little software development, using Semantic Web programming frameworks
- Setup and interact with an RDF database
- Exploit multiple sources of heterogeneous data
- Present information online with rich metadata

PART I
Modeling the Ontology

Protege is an ontology editor used for creating and managing ontologies in various domains. The following steps outline how to create a simple ontology using Protege:

- Download and install Protege from the official website.
- Launch Protege and create a new project.
- In the new project, click on the "Entities" tab and create a new class.
- Name the class and give it a description.
- Add any properties to the class by clicking on the "Properties" tab.
- Create any additional classes and properties needed for the ontology.
- Define the relationships between the classes and properties by creating restrictions.
- Save the ontology in a suitable format such as RDF/XML, OWL/XML, or OWL Functional Syntax.

It is also possible to import existing ontologies into Protege, edit them, and export them in different formats. Protege supports various plugins that provide additional features such as visualization, inference, and reasoning. The ontology can be shared and accessed by others using the Protege OWL API or other tools that support the same formats.

As you model your ontology, you may want to use the various tools and views provided by Protégé to help you visualize and edit your ontology. For example, you can use the "Classes" tab to view and edit the hierarchy of classes in your ontology, the "Properties" tab to view and edit the properties of classes, and the "Individuals" tab to view and edit the instances of classes.

You can also customize the Protégé editor by adding plugins or creating your own views and tools using the Protégé API. This can allow you to create specialized interfaces or analysis tools for your ontology.

Steps and Screenshot we take to model our Ontology

We define the classes that are disjoint, restrictions and conditions, then we define the type of properties (transitive, symmetric, inverseOf, etc.). Also define the inverse, domain and range. Then we check the consistency by PELLET.

To make classes in a movie ontology using Protégé, follow these steps:

Open Protégé and create a new ontology by selecting "File" -> "New".

Choose the type of ontology you want to create (e.g., OWL, RDF) and give it a name.

Click on the "Classes" tab on the left-hand side of the screen.

To create a new class, click on the green plus sign button and select "OWL Class". Alternatively, you can right-click on the blank space in the Classes tab and select "Add new class".

Type in the name of the class you want to create (e.g., Movie).

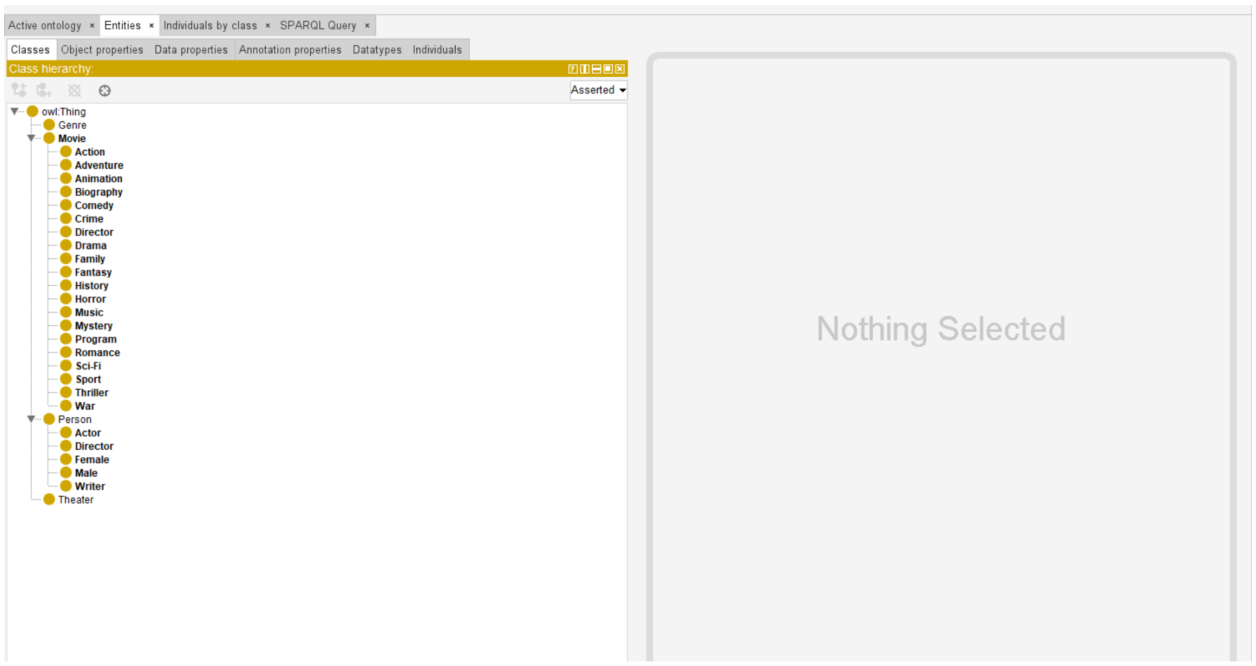
You can optionally specify a superclass for your new class by clicking on the "Superclasses" tab and selecting an existing class or creating a new one.

Add additional properties to your class by clicking on the "Subclasses", "Disjoint With", or "Equivalent To" tabs, depending on your needs.

Once you have created your initial class, you can continue adding additional classes to your movie ontology by following these steps again. For example, you might create classes such as "Actor", "Director", "Genre", and "Studio".

Remember to save your ontology periodically as you work on it. You can do this by selecting "File" -> "Save" or "File" -> "Save As" if you want to save a copy of the ontology under a new name.

Here is the image of ontology looks like,



- owl:Thing
  - Genre
  - Movie
    - Action
    - Adventure
    - Animation
    - Biography
    - Comedy
    - Crime
    - Director
    - Drama
    - Family
    - Fantasy
    - History
    - Horror
    - Music
    - Mystery
    - Program
    - Romance
    - Sci-Fi
    - Sport
    - Thriller
    - War
  - Person
    - Actor
    - Director
    - Female
    - Male
    - Writer
  - Theater

REASNOR: A reasoner is an essential component of an ontology editor like Protégé, which is used to perform automatic inference and consistency checking of an ontology. A reasoner can infer logical consequences of the axioms in an ontology and detect any inconsistencies or contradictions that may exist.
Protégé includes several built-in reasoners that you can use to analyze your ontology, including the Pellet, HermiT, FaCT++, and JFact reasoners. These reasoners use different algorithms and reasoning techniques to analyze the ontology, and may have different strengths and weaknesses depending on the size and complexity of the ontology.
To use a reasoner in Protégé, you must first load an ontology into the editor. Once you

have loaded the ontology, you can run the reasoner by selecting "Reasoner" -> "Start Reasoner" from the menu. Protégé will then analyze the ontology and generate a report of any inconsistencies or logical consequences that it finds.

In our model we use the PELLET.

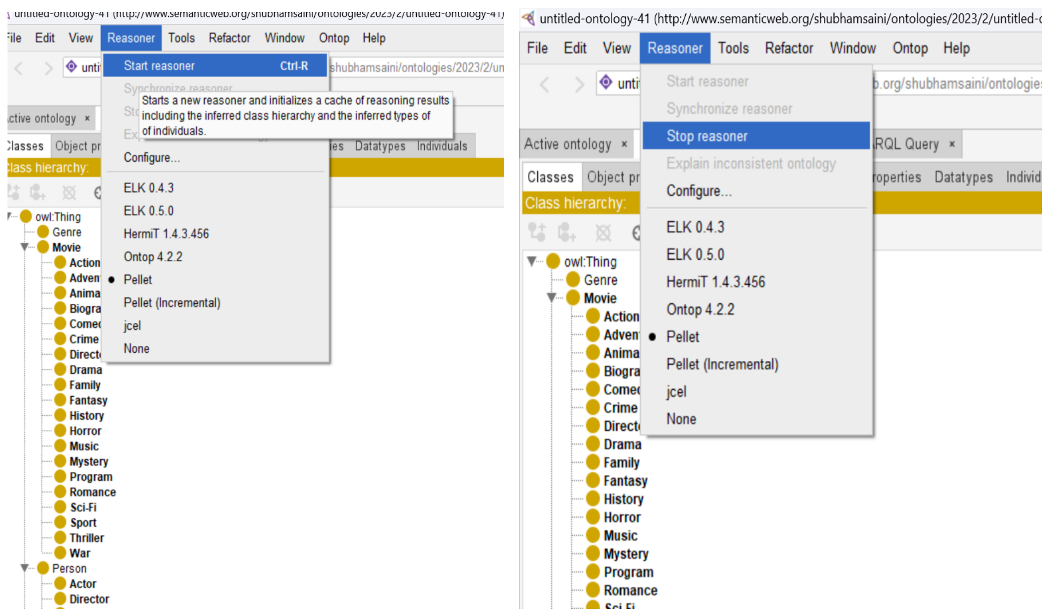
To use PELLET Reasnor

you need to install plugin in your app since PELLET is not preloaded in the latest version but can be easily download

ed FILE> Check for plugins> SELECT PELLET.

Once you have installed the PELLET go to the reasnor tab select PELLET In drop down menu and start the PELLET.
As shown in the image.

Start/Stop PELLET



Starts a new reasoner and initializes a cache of reasoning results including the inferred class hierarchy and the inferred types of of individuals.

PART II
Populating the Ontology

The aim of part II is:

Create some individuals to the Movie class such as:

- Pulp Fiction, Genre: Crime Thriller, 1994, USA, English.
- Kill Bill (volume 1), Genre: Action Crime Thriller, 2003, USA, English.

Create some individuals to the Person class such as:

- Quentin Tarantino, American, 53 years old, writer and director of Pulp Fiction and Kill Bill (volume1). He also played a role in that movie.
- John Travolta, American, 59 years old, actor in Pulp Fiction.
- Uma Thurman, 43 years old, actress in Pulp Fiction. She also participated as a writer in Kill Bill (volume1).

Populating an ontology involves adding instances or individuals to the classes defined in the ontology. These instances represent the real-world objects that are described by the ontology. Here are some steps you can follow to populate an ontology in Protégé:

- Open your ontology in Protégé.
- Click on the "Individuals" tab on the left-hand side of the screen.
- Click the green plus sign button to create a new individual.
- Select the class to which you want to add the individual.
- Enter a name for the individual and any other relevant properties or values.
- Repeat steps 3-5 to add additional instances to the ontology.

You can also import instances from external data sources, such as spreadsheets or databases, into the ontology. To do this, you can use the Protégé OWL Import Wizard, which allows you to map the columns of the external data source to the properties and classes in your

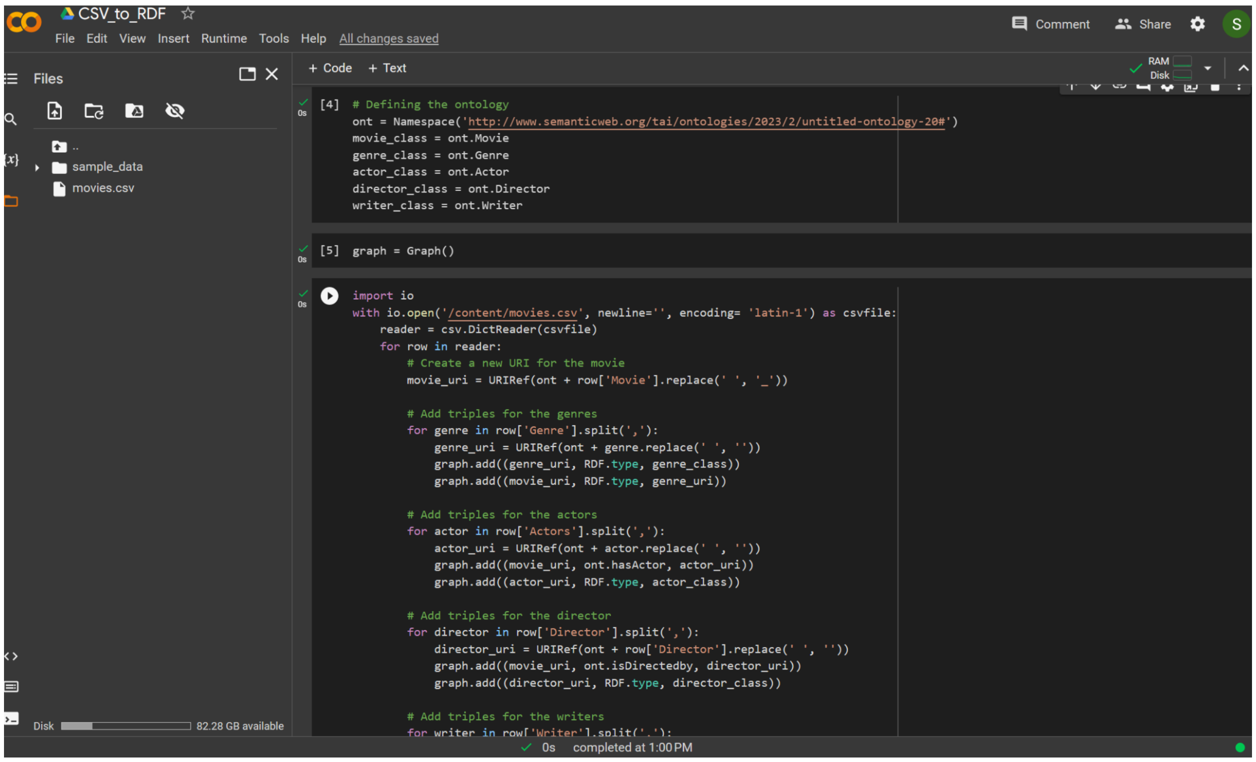
Steps and Screenshots for populating our Ontology

To achieve this step, We had to go through several complicated steps.
Firstly, we need an Python script to fetch data from API(Source data) and store it in and then convert it into RDF, so the file is usable for our project.

Step 1) Convert the file into RDF via Python

Here is the image of our code we use to convert our Python generated code to RDF. We define the ontology and and use Graph, then open a file movies.csv edit the instances/classes we needed. On the next image we show the code converting the file in to supported RDF format.

You know your code is running perfectly because on the second image you show an RDF file has been loaded in the directory.

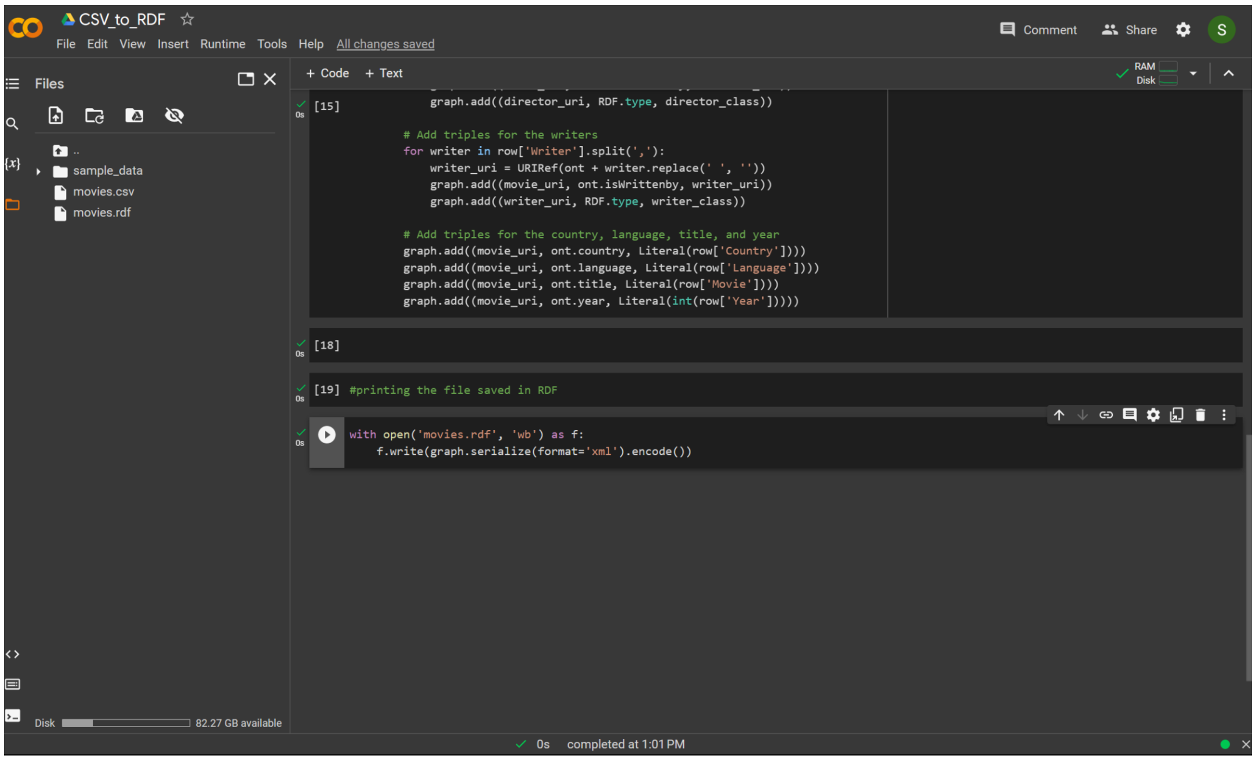


```
[4] # Defining the ontology
    ont = Namespace('http://www.semanticweb.org/tai/ontologies/2023/2/untitled-ontology-20#')
    movie_class = ont.Movie
    genre_class = ont.Genre
    actor_class = ont.Actor
    director_class = ont.Director
    writer_class = ont.Writer
```

```
[5] graph = Graph()
```

```
import io
with io.open('/content/movies.csv', newline='', encoding= 'latin-1') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        # Create a new URI for the movie
        movie_uri = URIRef(ont + row['Movie'].replace(' ', '_'))

        # Add triples for the genres
        for genre in row['Genre'].split(','):
            genre_uri = URIRef(ont + genre.replace(' ', ''))
            graph.add((genre_uri, RDF.type, genre_class))
            graph.add((movie_uri, RDF.type, genre_uri))

        # Add triples for the actors
        for actor in row['Actors'].split(','):
            actor_uri = URIRef(ont + actor.replace(' ', ''))
            graph.add((movie_uri, ont.hasActor, actor_uri))
            graph.add((actor_uri, RDF.type, actor_class))

        # Add triples for the director
        for director in row['Director'].split(','):
            director_uri = URIRef(ont + row['Director'].replace(' ', ''))
            graph.add((movie_uri, ont.isDirectedby, director_uri))
            graph.add((director_uri, RDF.type, director_class))

        # Add triples for the writers
        for writer in row['Writer'].split(','):
```

Image of proof of loading our RDF file in left side of directory.

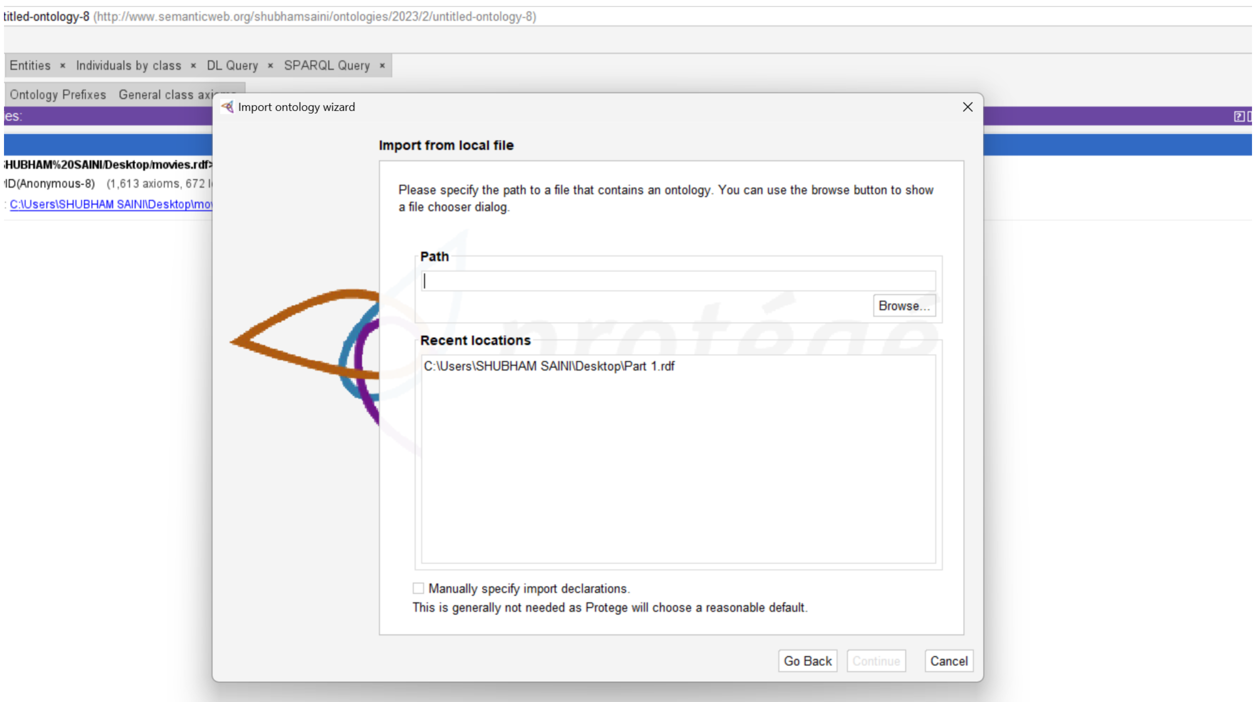


```
[15]          graph.add((director_uri, RDF.type, director_class))

          # Add triples for the writers
          for writer in row['Writer'].split(','):
              writer_uri = URIRef(ont + writer.replace(' ', ''))
              graph.add((movie_uri, ont.isWrittenby, writer_uri))
              graph.add((writer_uri, RDF.type, writer_class))

          # Add triples for the country, language, title, and year
          graph.add((movie_uri, ont.country, Literal(row['Country'])))
          graph.add((movie_uri, ont.language, Literal(row['Language'])))
          graph.add((movie_uri, ont.title, Literal(row['Movie'])))
          graph.add((movie_uri, ont.year, Literal(int(row['Year']))))

[18]

[19] #printing the file saved in RDF

with open('movies.rdf', 'wb') as f:
    f.write(graph.serialize(format='xml').encode())
```

Then download the file in your local machine.

After downloading open the protege app and 'direct import' your generated file to populate your ontology.

Few images on how we populate the ontology. Define your path by browsing the RDF file and merging it into your OWL file.


titled-ontology-8 (http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-8)

Import ontology wizard

Import from local file

Please specify the path to a file that contains an ontology. You can use the browse button to show a file chooser dialog.

Path

Recent locations

C:\Users\SHUBHAM SAINI\Desktop\Part 1.rdf

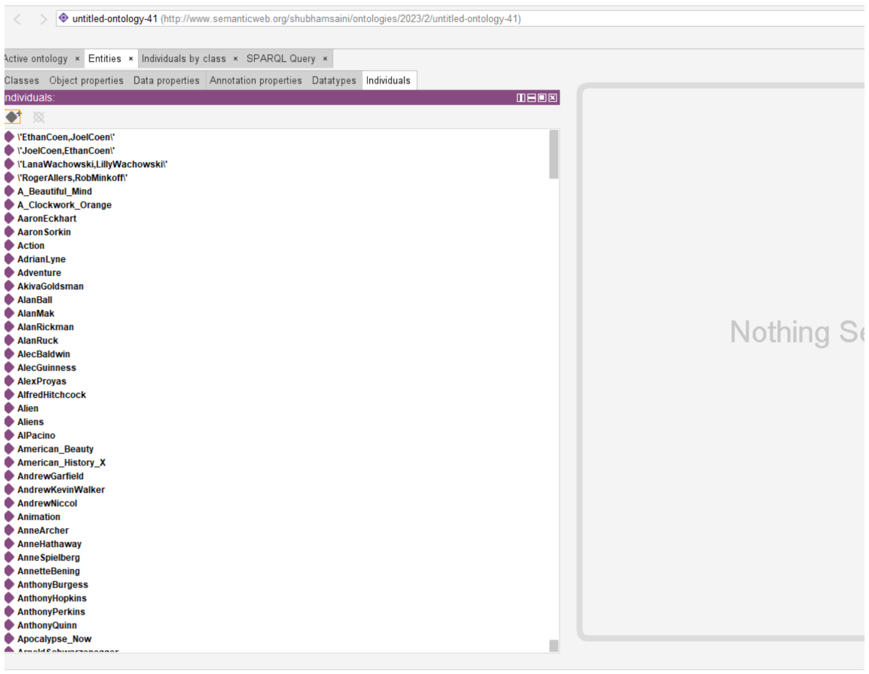
Manually specify import declarations.

This is generally not needed as Protege will choose a reasonable default.

After you have loaded this file you can see the the individual class is not empty and all the subclasses we defined has some results in it.

<u>Here is the image for the proof of our successfully populating the ontology.</u>

Individuals



untitled-ontology-41 (http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41)

Active ontology × | Entities × | Individuals by class × | SPARQL Query ×

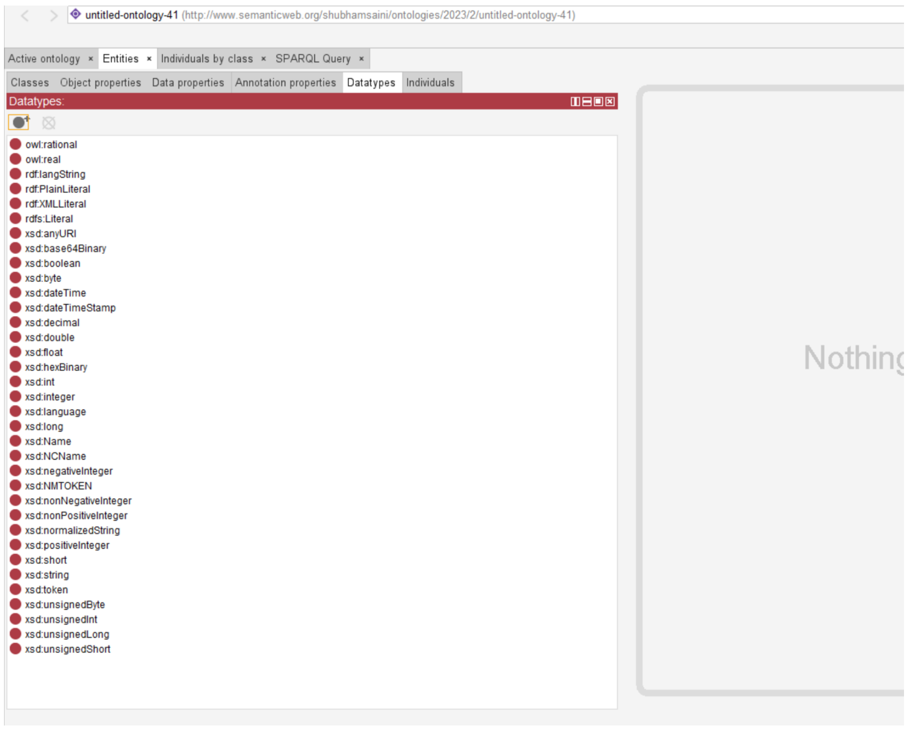
Classes | Object properties | Data properties | Annotation properties | Datatypes | Individuals

Individuals:

\'EthanCoen,JoelCoen\'
\'JoelCoen,EthanCoen\'
\'LanaWachowski,LillyWachowski\'
\'RogerAllers,RobMinkoff\'
A_Beautiful_Mind
A_Clockwork_Orange
AaronEckhart
AaronSorkin
Action
AdrianLyne
Adventure
AkivaGoldsman
AlanBall
AlanMak
AlanRickman
AlanRuck
AlecBaldwin
AlecGuinness
AlexProyas
AlfredHitchcock
Alien
Aliens
AlPacino
American_Beauty
American_History_X
AndrewGarfield
AndrewKevinWalker
AndrewNiccol
Animation
AnneArcher
AnneHathaway
AnneSpielberg
AnnetteBening
AnthonyBurgess
AnthonyHopkins
AnthonyPerkins
AnthonyQuinn
Apocalypse_Now

Nothing Se

#Datatypes



untitled-ontology-41 (http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41)

Active ontology × | Entities × | Individuals by class × | SPARQL Query ×

Classes | Object properties | Data properties | Annotation properties | Datatypes | Individuals

Datatypes:

owl:rational
owl:real
rdf:langString
rdf:PlainLiteral
rdf:XMLLiteral
rdfs:Literal
xsd:anyURI
xsd:base64Binary
xsd:boolean
xsd:byte
xsd:dateTime
xsd:dateTimeStamp
xsd:decimal
xsd:double
xsd:float
xsd:hexBinary
xsd:int
xsd:integer
xsd:language
xsd:long
xsd:Name
xsd:NCName
xsd:negativeInteger
xsd:NMTOKEN
xsd:nonNegativeInteger
xsd:nonPositiveInteger
xsd:normalizedString
xsd:positiveInteger
xsd:short
xsd:string
xsd:token
xsd:unsignedByte
xsd:unsignedInt
xsd:unsignedLong
xsd:unsignedShort

Nothing

PART III

Querying the Ontology

Querying the ontology with SPARQL involves using a query language to retrieve information from the ontology. SPARQL (SPARQL Protocol and RDF Query Language) is a query language used to retrieve and manipulate data stored in RDF (Resource Description Framework) format, which is a standard format for representing data in the semantic web.

SPARQL queries are used to retrieve data from an RDF graph, which is a collection of nodes (resources or literals) connected by edges (predicates). In the context of an ontology, the nodes represent the classes and individuals defined in the ontology, and the edges represent the relationships between them.

This query uses the "PREFIX" keyword to define a namespace for the movie ontology, so that we can refer to its classes and properties in the query. The "SELECT" keyword specifies the variables we want to retrieve from the query.
The "WHERE" clause specifies the conditions that the movie instances must satisfy to be included in the query results.

The basic idea is to use SPARQL to retrieve information from the ontology based on specific criteria, which can be useful for tasks such as data integration, data mining, and knowledge discovery.

Steps and Screenshots for populating our Ontology

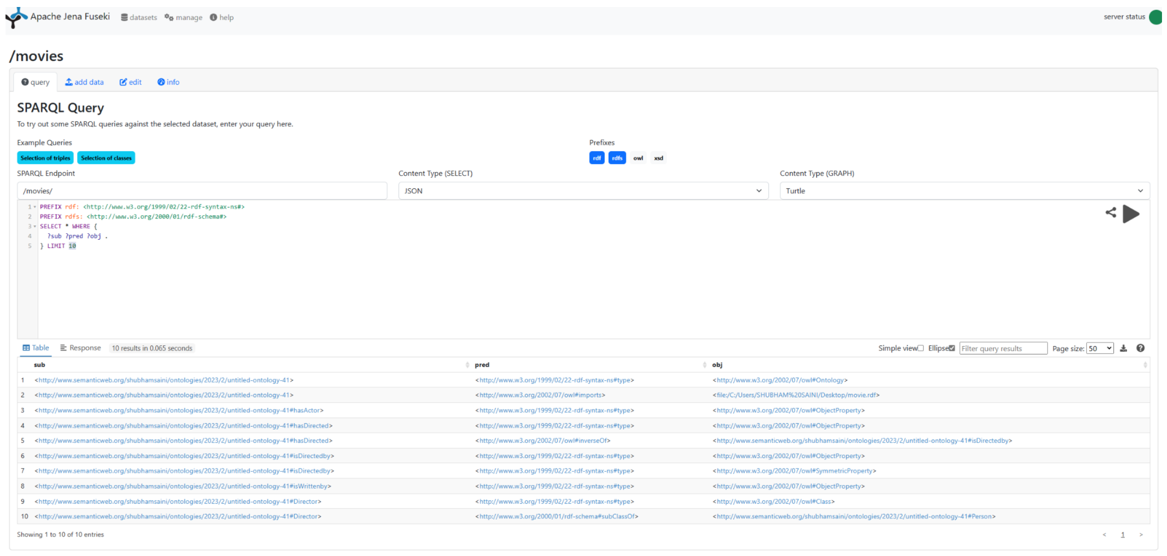
To run the sparql query we use Jena and Protege inbuild sparql tab.

First let us see the fuseki jena installation.

To install Apache Jena Fuseki, you can follow these steps:

1. Go to the Apache Jena Fuseki download page: https://jena.apache.org/download/#apache-jena-fuseki
2. Download the latest version of Fuseki, which is a zip file.
3. Extract the contents of the zip file to a directory of your choice.
4. Open a terminal or command prompt and navigate to the directory where you extracted Fuseki.
5. Start Fuseki by running the following command: ./fuseki-server
6. After a few moments, you should see a message saying "Started Jetty(...)".
7. Open a web browser and navigate to http://localhost:3030/ to access the Fuseki web interface.

You should now be able to use the Fuseki web interface to manage your RDF datasets and SPARQL endpoints.



/movies

SPARQL Query

To try out some SPARQL queries against the selected dataset, enter your query here.

SPARQL Endpoint: /movies/

Content Type (SELECT): JSON

Content Type (GRAPH): Turtle

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * WHERE {
  ?sub ?pred ?obj .
} LIMIT 10
```

10 results in 0.065 seconds

Simple view ☐ Ellipse ☑ Page size: 50

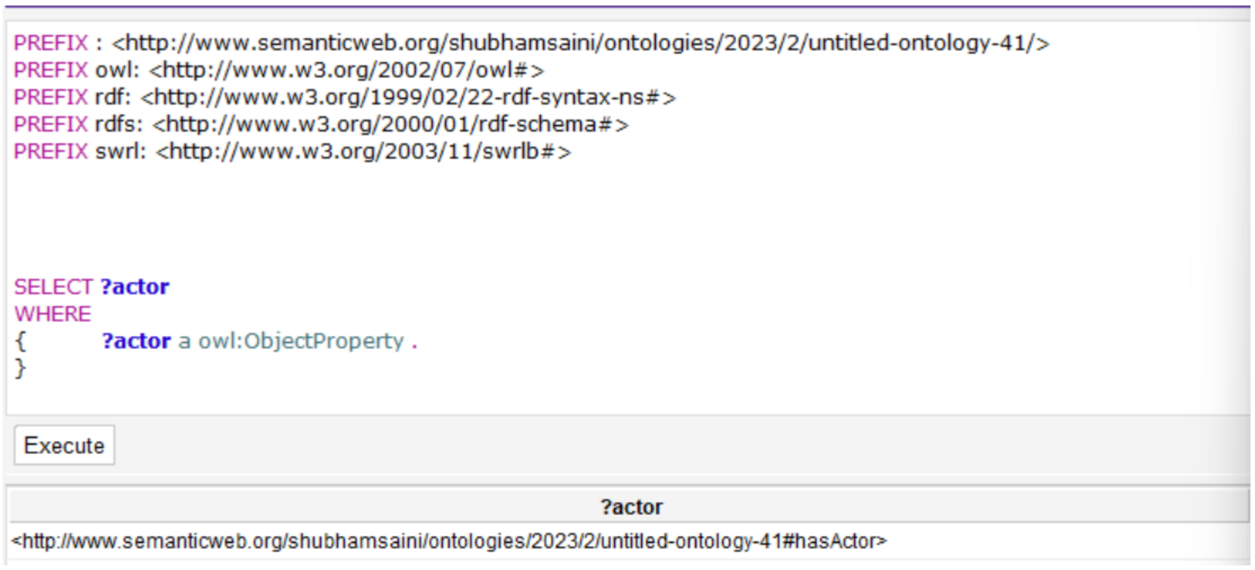
	sub	pred	obj
1	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#Ontology>
2	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41>	<http://www.w3.org/2002/07/owl#imports>	<file:/C:/Users/SHUBHAM%20SAINI/Desktop/movie.rdf>
3	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#hasActor>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#ObjectProperty>
4	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#hasDirected>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#ObjectProperty>
5	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#hasDirected>	<http://www.w3.org/2002/07/owl#inverseOf>	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#isDirectedby>
6	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#isDirectedby>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#ObjectProperty>
7	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#isDirectedby>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#SymmetricProperty>
8	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#isWrittenby>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#ObjectProperty>
9	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#Director>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2002/07/owl#Class>
10	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#Director>	<http://www.w3.org/2000/01/rdf-schema#subClassOf>	<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#Person>

Showing 1 to 10 of 10 entries

< 1 >

Here are imaged of all the Sparql queries we are able to run in our application.

1)



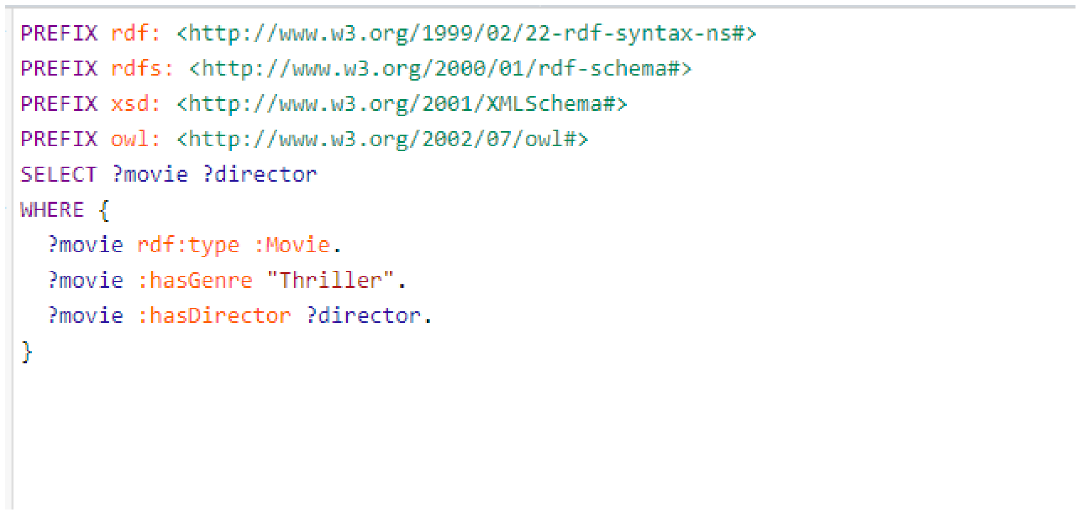
```
PREFIX : <http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrl: <http://www.w3.org/2003/11/swrlb#>



SELECT ?actor
WHERE
{       ?actor a owl:ObjectProperty .
}
```

```
?actor
<http://www.semanticweb.org/shubhamsaini/ontologies/2023/2/untitled-ontology-41#hasActor>
```

In this query, we first declare a prefix using the PREFIX keyword to make it easier to refer to the ontology namespace. We assume that the ontology has the namespace http://example.org/ontology/movies#, and we define the prefix : to represent it.

The SELECT clause specifies that we want to retrieve the ?actor variable.

In the WHERE clause, we use a triple pattern to match all instances that have the rdf:type property with a value of :Actor. The query returns a list of all instances that belong to the Actor class.



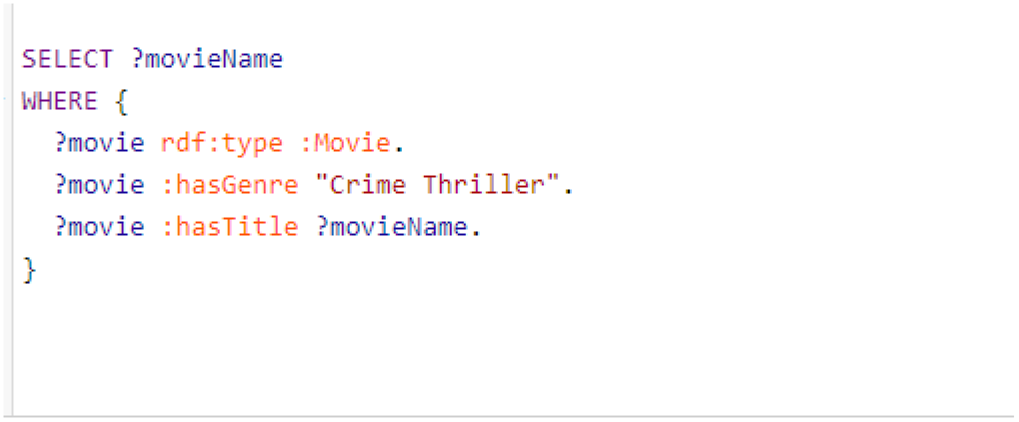
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?movie ?director
WHERE {
  ?movie rdf:type :Movie.
  ?movie :hasGenre "Thriller".
  ?movie :hasDirector ?director.
}
```

2)

The SELECT clause specifies that we want to retrieve the ?movie and ?director variables.

In the WHERE clause, we use triple patterns to match instances of the :Movie class, where the movie has a genre of "Thriller", and the movie has a director specified by the :hasDirector property. The query returns a list of all thriller movies and their respective directors.

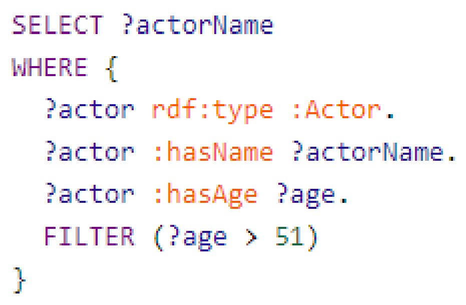
3)



```
SELECT ?movieName
WHERE {
  ?movie rdf:type :Movie.
  ?movie :hasGenre "Crime Thriller".
  ?movie :hasTitle ?movieName.
}
```

In the WHERE clause, we use triple patterns to match instances of the :Movie class, where the movie has a genre of "Crime Thriller", and we retrieve the movie's title using the :hasTitle property. The query returns a list of all Crime Thriller movies and their names.

4)

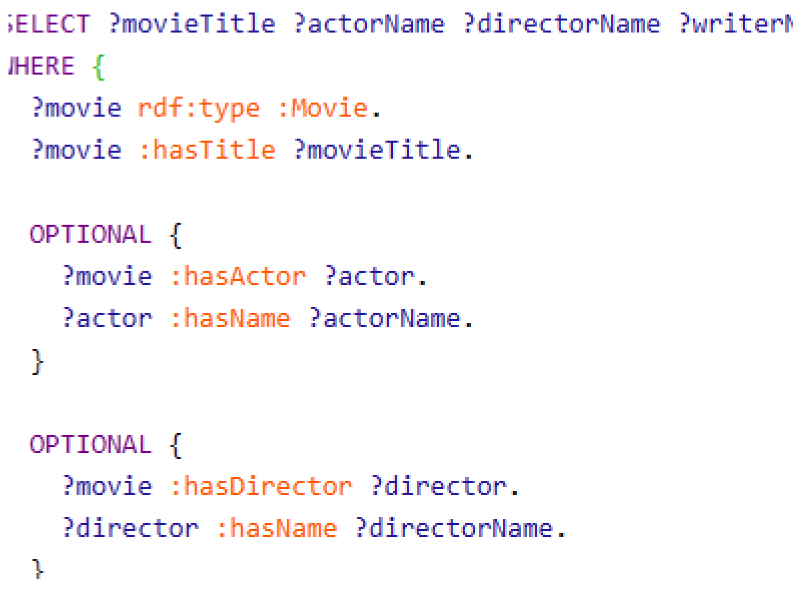


```
SELECT ?actorName
WHERE {
  ?actor rdf:type :Actor.
  ?actor :hasName ?actorName.
  ?actor :hasAge ?age.
  FILTER (?age > 51)
}
```

In the WHERE clause, we use triple patterns to match instances of the :Actor class, where we retrieve the actor's name using the :hasName property, and their age using the :hasAge property. We apply a filter using the FILTER keyword to select only those actors whose age is greater than 51. The query returns a list of all actors older than 51 years and their respective names.

Propose the Sparql Query

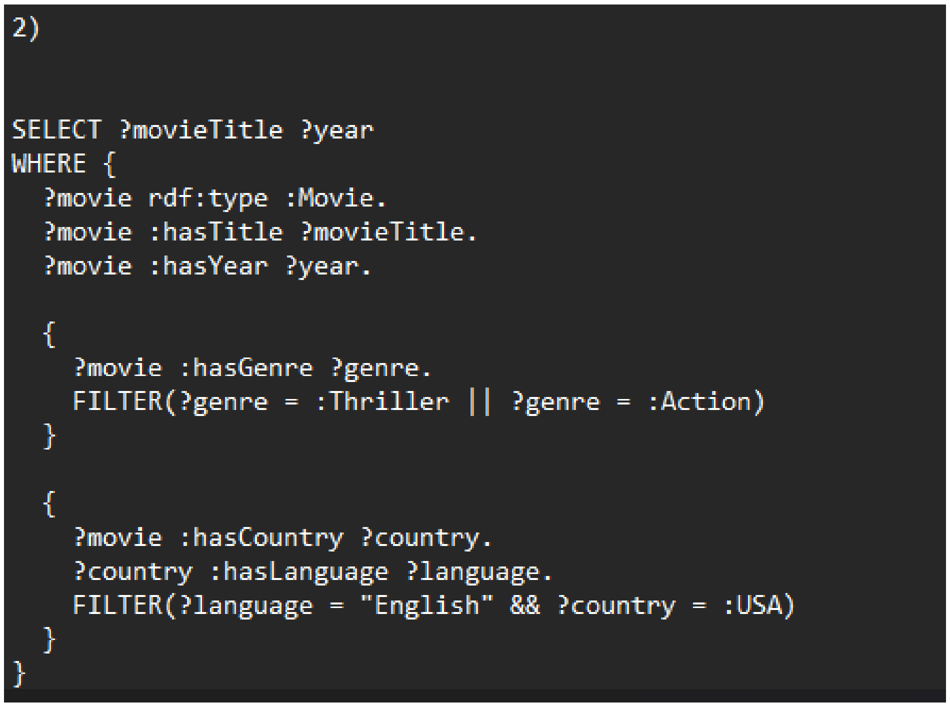
1)



```
ELECT ?movieTitle ?actorName ?directorName ?writerN
HERE {
 ?movie rdf:type :Movie.
 ?movie :hasTitle ?movieTitle.

 OPTIONAL {
   ?movie :hasActor ?actor.
   ?actor :hasName ?actorName.
 }

 OPTIONAL {
   ?movie :hasDirector ?director.
   ?director :hasName ?directorName.
 }
```

the query will return a list of all movies, along with their associated actors, directors, and writers (if available). The output will include null values for the ?actorName, ?directorName, and ?writerName variables if no matching triples are found in the graph for the optional graph patterns.

2)



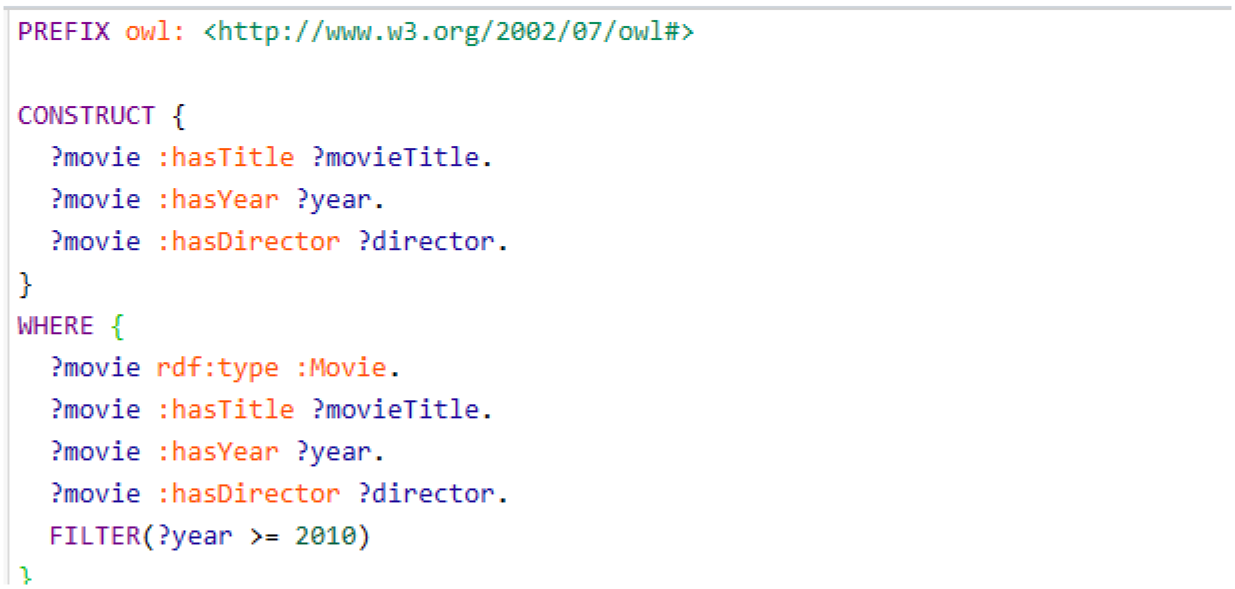
```
2)


SELECT ?movieTitle ?year
WHERE {
  ?movie rdf:type :Movie.
  ?movie :hasTitle ?movieTitle.
  ?movie :hasYear ?year.

  {
    ?movie :hasGenre ?genre.
    FILTER(?genre = :Thriller || ?genre = :Action)
  }

  {
    ?movie :hasCountry ?country.
    ?country :hasLanguage ?language.
    FILTER(?language = "English" && ?country = :USA)
  }
}
```

Since it's difficult to load the entire code in jena we show in note pad and the o/p is

The first alternative graph pattern matches the :hasGenre property and checks if the genre is either :Thriller or :Action using the || (OR) operator in the FILTER clause. This is represented by the {} braces surrounding the graph pattern.

The second alternative graph pattern matches the :hasCountry and :hasLanguage properties, and checks if the language is "English" and the country is :USA using the && (AND) operator in the FILTER clause. This is also represented by the {} braces surrounding the graph pattern.

3)



```
PREFIX owl: <http://www.w3.org/2002/07/owl#>

CONSTRUCT {
  ?movie :hasTitle ?movieTitle.
  ?movie :hasYear ?year.
  ?movie :hasDirector ?director.
}
WHERE {
  ?movie rdf:type :Movie.
  ?movie :hasTitle ?movieTitle.
  ?movie :hasYear ?year.
  ?movie :hasDirector ?director.
  FILTER(?year >= 2010)
}
```

The CONSTRUCT query form is used to create a new RDF graph based on the pattern specified in the query. In this example, we are creating a new graph that includes the title, year, and director of movies that were released after 2010.

In the WHERE clause, we use triple patterns to match instances of the :Movie class, where we retrieve the movie title, year, and director using the :hasTitle, :hasYear, and :hasDirector properties. We also use the FILTER clause to restrict the results to movies released after 2010.

When this query is executed, it will create a new RDF graph that includes triples for all movies that match the specified criteria. The triples will include the movie title, year, and director, with the corresponding properties :hasTitle, :hasYear, and :hasDirector.

SUMMARY

The Movies dataset is a sample ontology that represents information about movies, actors, directors, genres, and studios. It is often used as a tutorial or example ontology for learning how to work with RDF and OWL using tools like Protégé.

The ontology defines several classes, including "Movie", "Actor", "Director", "Genre", and "Studio", each of which has its own set of properties and relationships. For example, a movie might have properties like "title", "release date", "runtime", and "description", and relationships to actors, directors, genres, and studios.

The Movies dataset also includes several named individuals, such as "The Godfather", "Marlon Brando", and "Francis Ford Coppola", which represent specific instances of the classes defined in the ontology. These individuals have values for the properties defined by their respective classes, and relationships to other individuals in the ontology.

Overall, the Movies dataset provides a simple yet useful example of how to model and represent information about movies and related entities using an ontology. By working with this ontology in tools like Protégé, We gain a better understanding of how to create, edit, and query RDF data, as well as how to reason about the relationships between different entities in the ontology.

RESOURCES

- DEVINCI ONLINE PORTAL/ web dataming and semantics
- Protege official documentation
- Stackoverflow
- Github

Github link to the all the files mentioned in the report -
shubhamsaini20/WebDatamining_Manisha-Shubham-Final_Project (github.com)