KG Creation: Tabular Data

Credit

This assignment is identical to the one in the City uni course.

Tabular Data to Knowledge Graph (Task RDF)

Transform Pizza_data into RDF triples using your favorite programming language. Please document your code. Save the RDF data into turtle format (.ttl).

Subtask RDF.1 Discuss in the report the different transformation choices and how entity resolution was treated (30%). As we saw in the module, there is not a unique way to transform the elements in a table into elements of a knowledge graph (i.e., classes, object properties, data properties or instances). For example the column “menu item” contains the name of the pizza but it may also include additional information about toppings and type of pizza. Tip1: You will need to apply some basic text processing and entity recognition based on the ontology vocabulary. Tip2: After processing the data, one may also need to extend the ontology with new elements.

Subtask RDF.2 Create RDF triples (30%). Use the created ontology to guide the transformation e.g., to link the data to the ontology (e.g., via rdf:type) and to use the defined ontology properties (e.g., has_topping) and concepts (Margherita_Pizza).

Subtask RDF.3 For the cells in the columns city, country and state; instead of creating new URIs (e.g., new individuals) for the information in the table cells, reuse an entity URI from DBPedia, Wikidata or Google’s Knowledge Graph (http:// dbpedia.org/resource/Los_Angeles). Tip: communicate with their respective look-up services as we saw in the lab session (25%).

Subtask RDF.4 Correctness of the code and code documentation (15%).

Various data sets are used in the labs. Here are the links:

- Pizza Ontology
- SPARQL Playground
- Nobel Prize Data
- World cities dataset
- DBPedia
- OAEI datasets

In our initial approach to the project, we decided to create a hierarchical class structure. The classes would start with the country at the top, followed by the state as a subclass of the country, then the city as a subclass of the state, the restaurant as a subclass of the city, and finally, the pizza as a subclass of the restaurant, and so on. However, we quickly realized that this approach was not suitable for proper implementation.

We then considered another approach where the different types of pizzas and restaurants would be extracted from a given data table, enriching it with additional information as columns. This included columns such as pizza type and restaurant type. In this way, all the classes were separate entities, except for two cases where the different pizza types inherited from the pizza class and the different restaurant types inherited from the restaurant class. However, this modeling approach also proved to be unsustainable, as we discovered after starting the implementation.

We realized that it was not necessary to define the classes hierarchically, and what actually happened in practice was that the different types of pizzas could be related to the pizza_types class using the rdf:type relationship, which represents the pizza types. Each specific pizza sold by a restaurant could then be connected to the corresponding pizza type using the same relationship. This approach could also be applied to restaurants and different restaurant types. Other entities were also considered as separate classes in this approach.

Additionally, during the implementation phase, we realized that if the name of each pizza is not unique as an item in the menu, it could potentially have the same name as another pizza type. For example, Margherita pizza is a specific type of pizza, but it may also have exactly the same name as an item in the menu of a specific restaurant. This can lead to incorrect relationships, such as connecting multiple different prices to all restaurants with the same item name. This can result in inaccurate knowledge graph data.

To address this issue and ensure unique naming for each item in a restaurant's menu, we used the combination of the item name and the restaurant name. This resulted in a unique identifier for each item in the format of item_name + _of_ + restaurant_name. This approach guarantees that each item in every restaurant has a distinct and identifiable name.

Below are the implementation codes along with the presented results.

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv("/content/INM713_coursework_data_pizza_8358_1_reduced.csv")

# Check for NaN values in each column
nan_values = df.isnull().sum()
print("NaN values per column:")
print(nan_values)
```

```
# Explore the 'categories' column and extract unique category values
categories_column = df['categories']
categories = set()

# Iterate over each row in the 'categories' column
for row in categories_column:
    if isinstance(row, str):
        # Split the row by commas and add each category to the set
        row_categories = row.split(',')
        for category in row_categories:
            categories.add(category.strip())

# Print unique category values
print("Unique category values:")
for category in categories:
    print(category)
```

This code snippet performs the following tasks:

1. It imports the pandas module, which is used for data manipulation and analysis.

2. The provided CSV file, "INM713_coursework_data_pizza_8358_1_reduced.csv", is read and stored in a pandas DataFrame object, df.

3. It checks for NaN (null) values in each column of the DataFrame using the isnull() and sum() functions. The results, representing the number of NaN values per column, are stored in the nan_values variable.

4. The 'categories' column in the DataFrame is explored to extract unique category values.

5. A set, categories, is initialized to store the unique category values.

6. The code iterates over each row in the 'categories' column.

7. If the row is a non-null string value, it splits the row by commas and adds each category to the set after stripping any leading or trailing whitespace.

8. Finally, the unique category values are printed out.

In this stage, it was determined that there are 65 records missing postal code values. In the subsequent queries, their names were extracted. However, it should be noted that these 65 records belong to menu items of restaurants that do not have a postal code. The number of restaurants may not necessarily be 65, and it is likely to be significantly lower because each restaurant may have multiple items in the menu.

NaN values per column:

name	0
address	0
city	0
country	0
postcode	65
state	0
categories	0
menu item	0
item value	562
currency	559
item description	1984

Top 10 most frequent categories:

Pizza Place	1587
Restaurants	1120
Restaurant	884
Pizza	827
Italian Restaurant	551
Caterers	383
Italian Restaurants	370
Take Out Restaurants	350
American Restaurant	339
Bar	184
Pizza Place and Italian Restaurant	110
Mediterranean Restaurants	84
Sports Bar	83
Sandwich Shops	77
American Restaurants	76
Delicatessens	59
American Restaurant and Italian Restaurant	55
Bars	53
Family Style Restaurants	51
Food & Entertainment	51
Pizza Restaurants	50
Mexican Restaurant	47
Food and Beverage	45
Catering	42
Bar & Grills	42

dtype: int64

In the above image, the word cloud displays the most frequently occurring words in restaurant menus. These insights were used to extract different types of pizzas.

```
Top 10 pizza types:
Other                2548
Cheese                374
Margherita            115
Pepperoni             103
Hawaiian               75
Veggie                 75
Bbq Chicken            73
Buffalo Chicken        57
Greek                  41
Supreme                25
Name: 0, dtype: int64
```

In the next stage, text analysis was performed on the specifications of the pizza items in the restaurants' menus. More than 50 types of pizza were extracted, and a new column was added to the table to indicate the type of each

pizza item in a restaurant. This process involved several steps to classify the 2500 unidentified pizza types into approximately 2300 categorized pizzas and 1300 uncategorized pizzas, reducing the total number of pizza types.

	pizza_type	count
32	other	1294
10	cheese	306
23	margherita	135
55	veggie	131
57	white	117
11	chicken	111

In the end, the table was transformed as follows:

	name	address	city	country	postcode	state	categories	menu item	item value	currency	item description	pizza_type
0	Little Pizza Paradise	Cascade Village Mall Across From Target	Bend	US	97701	OR	Pizza Place	Bianca Pizza	22.50	USD	NaN	bianca
1	Little Pizza Paradise	Cascade Village Mall Across From Target	Bend	US	97701	OR	Pizza Place	Cheese Pizza	18.95	USD	NaN	cheese
2	The Brentwood	148 S Barrington Ave	Los Angeles	US	90049	Brentwood	American Restaurant,Bar,Bakery	Pizza, Margherita	12.00	USD	NaN	margherita
3	The Brentwood	148 S Barrington Ave	Los Angeles	US	90049	Brentwood	American Restaurant,Bar,Bakery	Pizza, Mushroom	13.00	USD	NaN	mushroom
4	The Brentwood	148 S Barrington Ave	Los Angeles	US	90049	Brentwood	American Restaurant,Bar,Bakery	Pizza, Puttenesca	13.00	USD	Olives, onions, capers, tomatoes	other

In the next step, an attempt was made to extract the triplets from the table. However, due to the changes made in the names of restaurant menu items, these triplets were not used.

subject	predicate	object
Little Pizza Paradise	address	Cascade Village Mall Across From Target
The Brentwood	address	148 S Barrington Ave
Bravo Pizza Hollywood	address	5142 Hollywood Blvd
Lucky's Pub	address	801 Saint Emanuel St
Roadhouse Cafe	address	478 South St
...	...	...
Supreme Pizza	pizza_type	supreme

Now, after performing the necessary processing on the SPARQLWrapper library table, rdflib was installed to use them for knowledge graph construction.

As requested, well-known and popular knowledge graphs were used to extract entities related to countries, cities, and states. The following code was written for this purpose.

Searching for entities in famous knowledge bases:

```
import pandas as pd
from rdflib import Graph, URIRef, RDF, Literal, XSD, Namespace
from SPARQLWrapper import SPARQLWrapper, JSON
import requests

# function to get entity URI from DBPedia
def get_dbpedia_uri(entity):
    uri = ""
    sparql = SPARQLWrapper("http://dbpedia.org/sparql")
    query = """select ?s where { ?s rdfs:label "%s"@en } limit 1""" % entity
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()
    for result in results["results"]["bindings"]:
        uri = result["s"]["value"]
    return uri

# function to get entity URI from Wikidata
def get_wikidata_uri(entity):
    uri = ""
    sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
    query = """select ?s where { ?s skos:altLabel "%s"@en } limit 1""" % entity
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()
    for result in results["results"]["bindings"]:
        uri = result["s"]["value"]
    return uri

# function to get entity URI from Google's Knowledge Graph
def get_googlekg_uri(entity):
    uri = ""
    api_key = "YOUR_API_KEY"  # replace with your own API key
    service_url = "https://kgsearch.googleapis.com/v1/entities:search"
    params = {
        "query": entity,
        "key": api_key,
```

```
        "limit": 1,
        "indent": True
    }
    url = service_url + "?" + "&".join("%s=%s" % (k, v) for k, v in
params.items())
    response = requests.get(url).json()
    for element in response["itemListElement"]:
        uri = element["result"]["@id"]
    return uri
```

This code imports necessary libraries such as pandas for data manipulation, rdflib for working with RDF graphs, SPARQLWrapper for querying SPARQL endpoints, and requests for making HTTP requests.

The code also defines three functions:

1. `get_dbpedia_uri`: This function takes an `entity` parameter and uses the SPARQLWrapper library to query the DBpedia SPARQL endpoint for the URI of that entity's label. It returns the URI of the entity.

2. `get_wikidata_uri`: This function takes an `entity` parameter and uses the SPARQLWrapper library to query the Wikidata SPARQL endpoint for the URI of that entity's alternative label. It returns the URI of the entity.

3. `get_googlekg_uri`: This function takes an `entity` parameter and makes a request to Google's Knowledge Graph API to search for the entity. It uses an API key and constructs the necessary parameters for the request. It then parses the response to extract the URI of the entity and returns it.

These functions are used to retrieve the URIs of entities from different knowledge bases (DBpedia, Wikidata, and Google's Knowledge Graph) based on their labels or alternative labels.

Now, we move on to the relationships between countries and states. We introduce countries and states as entities of type "Country" and "State" respectively using the predicate `rdf:type`. Then, we use the found URIs for them from well-known knowledge bases and introduce them as URIs in the knowledge graph. We consider the relationship between the country and state accordingly.

```
import time
import pandas as pd
from rdflib import Graph, URIRef, RDF, Literal, XSD, Namespace, OWL
from SPARQLWrapper import SPARQLWrapper, JSON
import requests
from urllib.error import HTTPError
import urllib.parse #for parsing strings to URI's

g = Graph()
country = df[['state', 'country']]
```

```
country = country.dropna()
country = country.drop_duplicates()
pizza_type=df[['name', 'menu item','pizza_type']]
currency=df[['name', 'menu item','item value','currency']]
price=df[['name', 'menu item','item value']]
e=Namespace('http://www.city.ac.uk/ds/inm713/hadi_ghasemi/relations/')
n=Namespace('http://www.city.ac.uk/ds/inm713/hadi_ghasemi/')

def search_entity(entity):
    max_retries = 3
    retries = 0
    while retries < max_retries:
        try:
            uri = get_dbpedia_uri(entity)
            if uri:
                return uri
            time.sleep(1)  # add a delay to avoid too many requests
            uri = get_wikidata_uri(entity)
            if uri:
                return uri
            return None
        except HTTPError as ex:
            if ex.code == 429:
                print("Too many requests. Retrying in 10 seconds...")
                time.sleep(10)  # add a longer delay before retrying
                retries += 1
            else:
                raise
#(("relationshipB", rdflib.OWL.Class),
#("ClassD", rdflib.OWL.DatatypeProperty),
#("attributeD", rdflib.OWL.ObjectProperty)):

uri_dict={}
edge = URIRef(e + "state_country")
cnt=URIRef(n + "countries")
stts=URIRef(n + "states")
Cname=URIRef(e + "cname")
Sname=URIRef(e + "sname")

g.add((edge, RDF.type,OWL.ObjectProperty))
g.add((Cname, RDF.type,OWL.DatatypeProperty))
g.add((Sname, RDF.type,OWL.DatatypeProperty))



# Modify the loop to use the search_entity() function
```

```
for index, row in country.iterrows():
    print(row["state"])
    State = row["state"]
    Country = row["country"]
    if Country not in uri_dict:
        uri_dict[Country] = search_entity(Country)
    if State not in uri_dict:
        uri_dict[State] = search_entity(State)
        if uri_dict[State] is None:
          uri_dict[State]=URIRef(n +"states/"+ urllib.parse.quote(row["state"]))
        print(uri_dict[State])

    if uri_dict[Country] is not None:
        target_uri = URIRef(uri_dict[Country])
        if uri_dict[State] is not None:
            print("+")
            source_uri = URIRef(uri_dict[State])


            g.add((target_uri,RDF.type,cnt))
            g.add((source_uri,RDF.type,stts))
            g.add((source_uri, edge, target_uri))

            g.add((target_uri,Cname,Literal(Country, datatype=XSD.string)))
            g.add((source_uri,Sname,Literal(State, datatype=XSD.string)))
```

This code performs the following tasks:

1. Imports the necessary libraries, including time, pandas, rdflib, SPARQLWrapper, requests, and urllib.

2. Creates an RDF graph object using rdflib.Graph().

3. Processes the 'country' dataframe to extract unique pairs of states and countries, removing any NaN values and duplicates.

4. Defines namespaces using rdflib.Namespace to create URIs for relationships and entities.

5. Defines three functions: 'search_entity' to search for URIs of entities from DBpedia and Wikidata, and 'get_dbpedia_uri' and 'get_wikidata_uri' which are called by 'search_entity' to perform the SPARQL queries.

6. Defines the URI references for relationship properties and datatype properties using rdflib.URIRef.

7. Adds RDF triples to the graph using the g.add() method. These triples define the class types and properties of the URIs, such as the relationship between states and countries, the names of countries, and the names of states.

8. Iterates through the 'country' dataframe, calling the 'search_entity' function to retrieve the URIs of the country and state entities. If a URI is not found, a new URI is created using the state name. The resulting URIs and their relationships are added to the graph.

9. Adds the country and state names as literals to their respective URIs using g.add().

10. The code does not include the implementation for the relationship between restaurants and cities, which would be similar to the country-state relationship in terms of retrieving entity URIs and adding relationships to the graph.

Overall, this code builds an RDF graph by extracting unique pairs of states and countries, searching for their URIs, and defining relationships and properties between them.

Then, we will repeat the same process for the relationship between cities and states, and then for the relationship between restaurants and cities.

In the next step, we connect the address of each restaurant as a literal to the respective restaurant.

```
edge = URIRef(e + "address")
Rname=URIRef(e + "rname")

g.add((edge, RDF.type,OWL.DatatypeProperty))
g.add((Rname, RDF.type,OWL.DatatypeProperty))


for index, row in address.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))
    target = Literal(row["address"], datatype=XSD.string)
    name=Literal(row["name"], datatype=XSD.string)


    #      restu  addr   literal
    g.add((source,edge, target))
    #     rest    name    literal
    g.add((source,Rname, name))
    #     rest    type      restaurant

edge = URIRef(e + "postcode")

g.add((edge, RDF.type,OWL.DatatypeProperty))
```

This code is part of a loop that adds triples to an RDF graph using the rdflib library. Let's break down what it does:

1. `edge = URIRef(e + "address")`: This line creates a URI reference for the property "address" by concatenating the base URI "e" with the string "address". It is used to represent the relationship between a restaurant and its address.

2. `Rname=URIRef(e + "rname")`: This line creates a URI reference for the property "rname" by concatenating the base URI "e" with the string "rname". It is used to represent the relationship between a restaurant and its name.

3. `g.add((edge, RDF.type, OWL.DatatypeProperty))` and `g.add((Rname, RDF.type, OWL.DatatypeProperty))`: These lines add triples to the RDF graph "g" to define the properties "edge" and "Rname" as datatype properties using the OWL vocabulary.

4. The loop iterates through each row in the "address" DataFrame. For each row, it performs the following steps:

   - `source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))`: This line creates a URI reference for the restaurant by concatenating the base URI "n" with the string "food_store/" and the URL-encoded version of the restaurant's name. It represents the restaurant's identity in the graph.

   - `target = Literal(row["address"], datatype=XSD.string)`: This line creates a literal value representing the address of the restaurant. The literal is associated with the datatype XSD.string.

   - `name = Literal(row["name"], datatype=XSD.string)`: This line creates a literal value representing the name of the restaurant. The literal is associated with the datatype XSD.string.

   - `g.add((source, edge, target))`: This line adds a triple to the RDF graph, stating that the restaurant (source) has an address (edge) with the given literal value (target).

   - `g.add((source, Rname, name))`: This line adds a triple to the RDF graph, stating that the restaurant (source) has a name (Rname) with the given literal value (name).

5. `edge = URIRef(e + "postcode")`: This line creates a new URI reference for the property "postcode" by concatenating the base URI "e" with the string "postcode". It will be used to represent the relationship between a city and its postcode.

6. `g.add((edge, RDF.type, OWL.DatatypeProperty))`: This line adds a triple to the RDF graph, stating that the property "postcode" is a datatype property using the OWL vocabulary.

Overall, this code is responsible for adding triples to the RDF graph to represent relationships between restaurants and their addresses, names, and between cities and their postcodes.

```
for index, row in postcode.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))
    target = Literal(row["postcode"], datatype=XSD.string)

    #       rest post   literal
    g.add((source,edge, target))

edge = URIRef(e + "category")

g.add((edge, RDF.type,OWL.ObjectProperty))
```

Then, we will repeat the same process for the relationship between restaurants and their postcode.

Next, we define restaurant categories as entities and then connect to the respective restaurants as a object properties.

```
edge = URIRef(e + "category")

g.add((edge, RDF.type,OWL.ObjectProperty))

for index, row in new_Categories.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))
    target = URIRef(n +"rcategories/"+ urllib.parse.quote(row["category"]))

    Category=URIRef(n+"restaurant")
    #      category  type   caregory
    g.add((target,RDF.type, Category))
    #      rest   catg   category
    g.add((source,edge, target))
```

This code below is another part of a loop that adds triples to an RDF graph. Let's break down what it does:

1. `edge = URIRef(e + "menu")`: This line creates a URI reference for the property "menu" by concatenating the base URI "e" with the string "menu". It is used to represent the relationship between a restaurant and its menu.

2. `Pname = URIRef(e + "pname")`: This line creates a URI reference for the property "pname" by concatenating the base URI "e" with the string "pname". It is used to represent the relationship between a menu item and its name.

3. `Pizza = URIRef(n + "food/pizza")`: This line creates a URI reference for the class "pizza" by concatenating the base URI "n" with the string "food/pizza". It is used to represent the type of a pizza menu item.

4. `g.add((edge, RDF.type, OWL.ObjectProperty))` and `g.add((Pname, RDF.type, OWL.DatatypeProperty))`: These lines add triples to the RDF graph, stating that the properties "edge" and "Pname" are an object property and a datatype property, respectively, using the OWL vocabulary.

5. The loop iterates through each row in the "pizza" DataFrame. For each row, it performs the following steps:

- `source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))`: This line creates a URI reference for the restaurant by concatenating the base URI "n" with the string "food_store/" and the URL-encoded version of the restaurant's name. It represents the restaurant's identity in the graph.

- target = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"]) +"_of_"+urllib.parse.quote(row["name"]))`: This line creates a URI reference for the menu item by concatenating the base URI "n" with the string "food/", the URL-encoded version of the menu item's name, "_of_", and the URL-encoded version of the restaurant's name. It represents the identity of the menu item in the graph.

- `name = Literal(row["menu item"], datatype=XSD.string)`: This line creates a literal value representing the name of the menu item. The literal is associated with the datatype XSD.string.

- `g.add((target, RDF.type, Pizza))`: This line adds a triple to the RDF graph, stating that the menu item (target) belongs to the class "pizza" (Pizza).

- `g.add((source, edge, target))`: This line adds a triple to the RDF graph, stating that the restaurant (source) has a menu item (edge) represented by the menu item's URI (target).

- `g.add((target, Pname, name))`: This line adds a triple to the RDF graph, stating that the menu item (target) has a name (Pname) with the given literal value (name).

```
edge = URIRef(e + "menu")
Pname=URIRef(e + "pname")
Pizza=URIRef(n+"food/pizza")

g.add((edge, RDF.type,OWL.ObjectProperty))
g.add((Pname, RDF.type,OWL.DatatypeProperty))

for index, row in pizza.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food_store/"+ urllib.parse.quote(row["name"]))
    target = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"])
+"_of_"+urllib.parse.quote(row["name"]))
```

```
    name=Literal(row["menu item"], datatype=XSD.string)

    #     pizza   type   pizza
    g.add((target,RDF.type, Pizza))
    #     rest    menu    pizza
    g.add((source,edge, target))
    #     pizza   name   literal
    g.add((target,Pname, name))
```

Overall, this code is responsible for adding triples to the RDF graph to represent the relationship between restaurants and their menu items, as well as the corresponding names of the menu items. Additionally, it defines the class "pizza" and assigns it as the type of pizza menu items.

In the next step, we add the prices of pizzas, along with their currency units, to the pizza entities. The code for this is provided below.

```
edge = URIRef(e + "value")

g.add((edge, RDF.type,OWL.DatatypeProperty))

for index, row in price.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"])
+"_of_"+urllib.parse.quote(row["name"]))
    target = Literal(row["item value"], datatype=XSD.float)

    #     pizza   price   value
    g.add((source,edge, target))

edge = URIRef(e + "value")

g.add((edge, RDF.type,OWL.DatatypeProperty))


for index, row in currency.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"])
+"_of_"+urllib.parse.quote(row["name"]))
    #target = URIRef(n +"currency/"+ row["currency"])
    target = Literal(row["currency"], datatype=XSD.string)
    #      pizza   currency  value
    g.add((source,edge, target))
```

Finally, relationships related to pizza types and descriptions are defined and added to the knowledge graph.

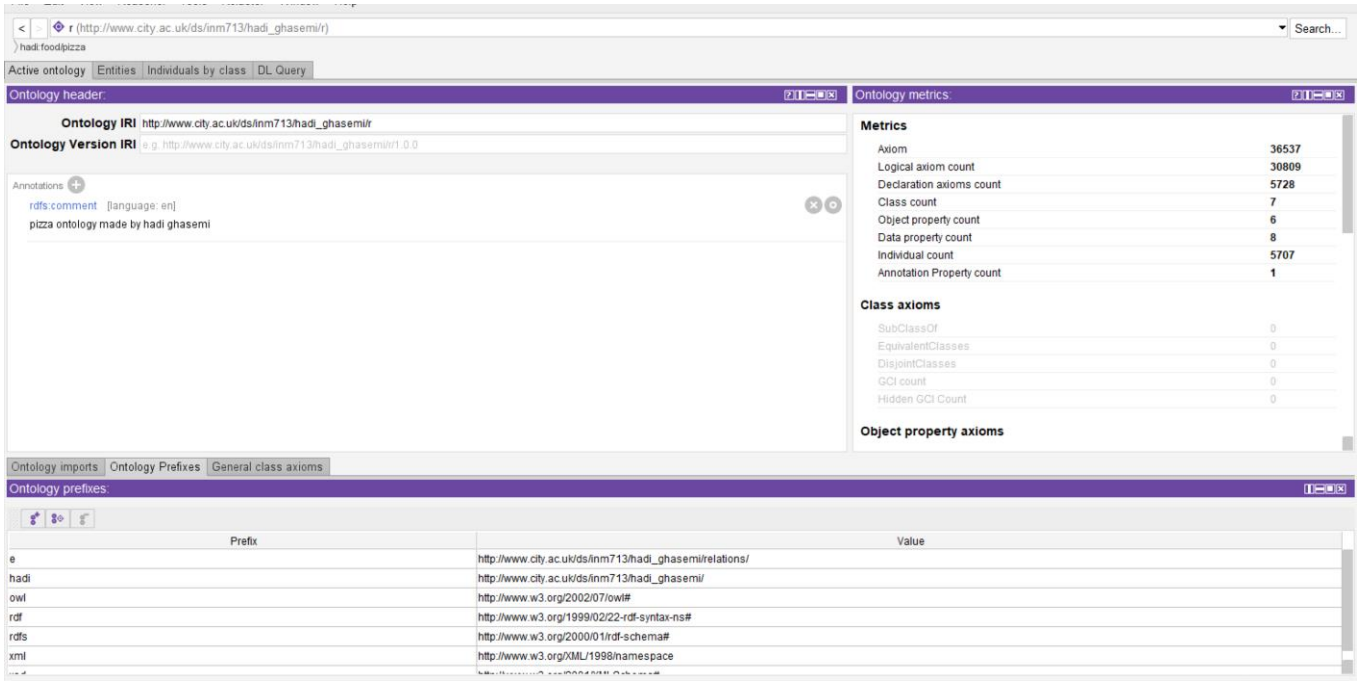
```
edge = URIRef(e + "pizza_type")
pizza_types=URIRef(n+"pizza_types")

g.add((edge, RDF.type,OWL.ObjectProperty))

for index, row in pizza_type.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"])
+"_of_"+urllib.parse.quote(row["name"]))
    target = URIRef(n +"pizza_type/" +urllib.parse.quote(row["pizza_type"]) )

    #     piza_type type  pizza_types
    g.add((target,RDF.type, pizza_types))
    #      pizza   type  pizza_type
    g.add((source,edge, target))
    #print(target)


edge = URIRef(e + "description")

g.add((edge, RDF.type,OWL.DatatypeProperty))


for index, row in item_description.iterrows():
    # add triple to rdf-graph
    source = URIRef(n +"food/"+ urllib.parse.quote(row["menu item"])
+"_of_"+urllib.parse.quote(row["name"]))
    target = Literal(row["item description"], datatype=XSD.string)

    #      pizza   desc   description
    g.add((source,edge, target))
```

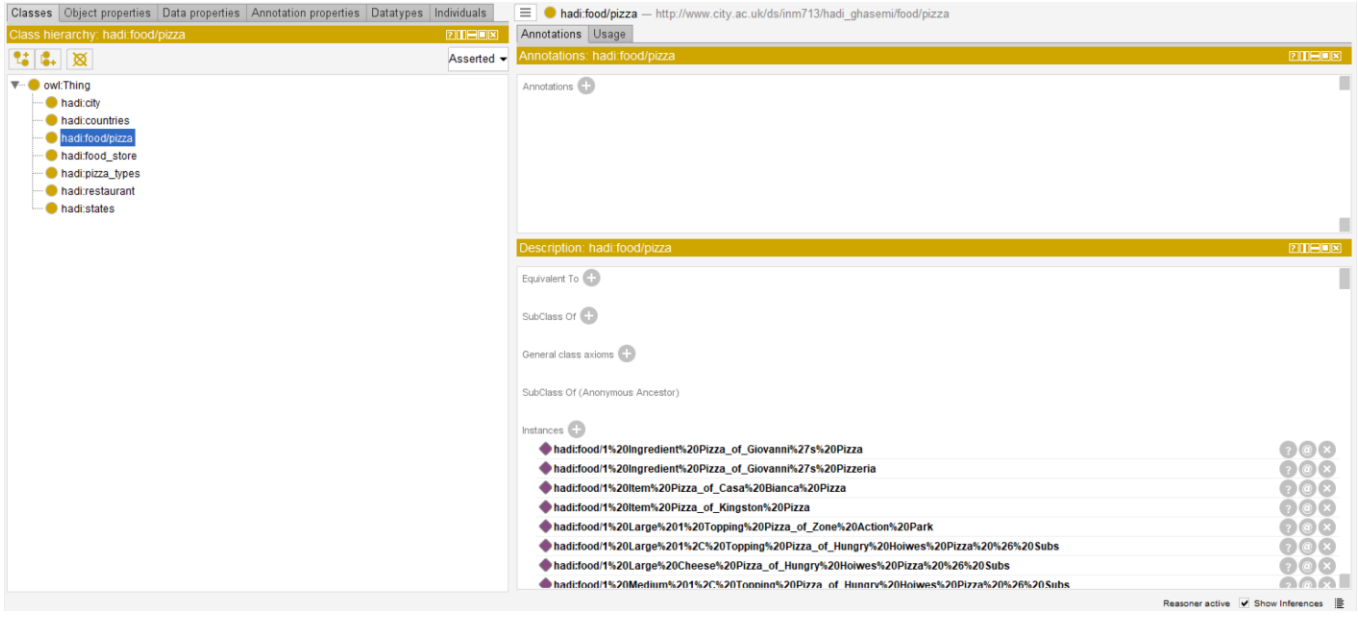
Finally, the created knowledge graph is saved in two formats: .owl and .ttl.

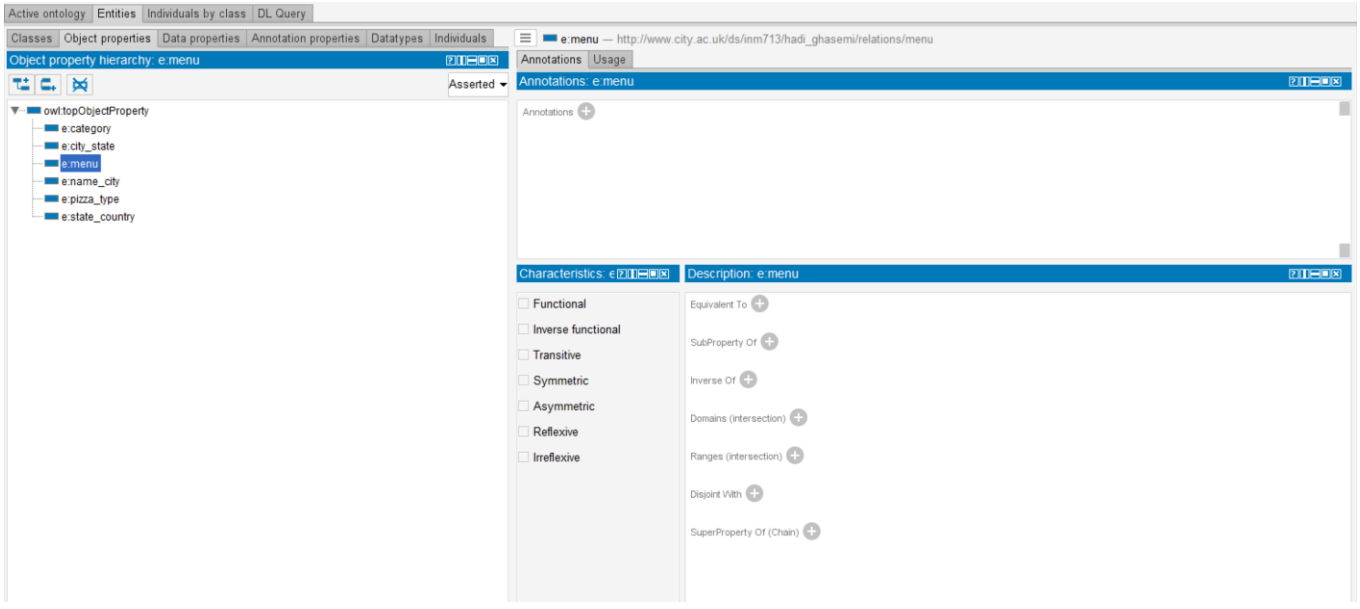
```
g.serialize(destination='/content/output.owl', format='xml')
g.serialize(destination='/content/output1.ttl', format='turtle')
```

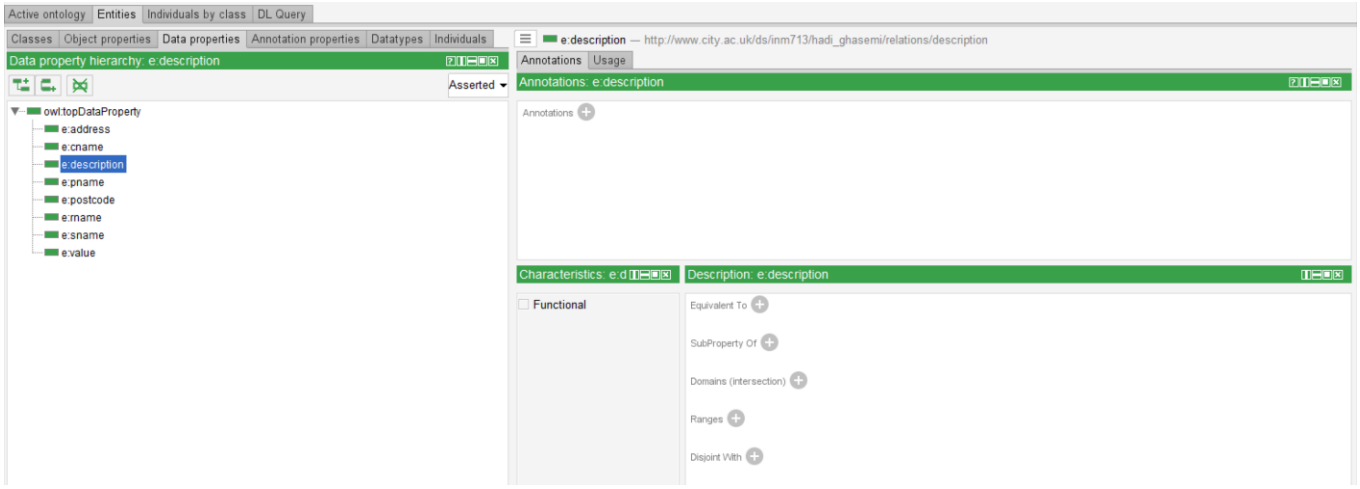
This means that the RDF graph, which represents the knowledge graph, is saved in the OWL (Web Ontology Language) format with the .owl file extension, and also in the Turtle (Terse RDF Triple Language) format with the .ttl file extension. These formats are commonly used for representing and storing semantic data.

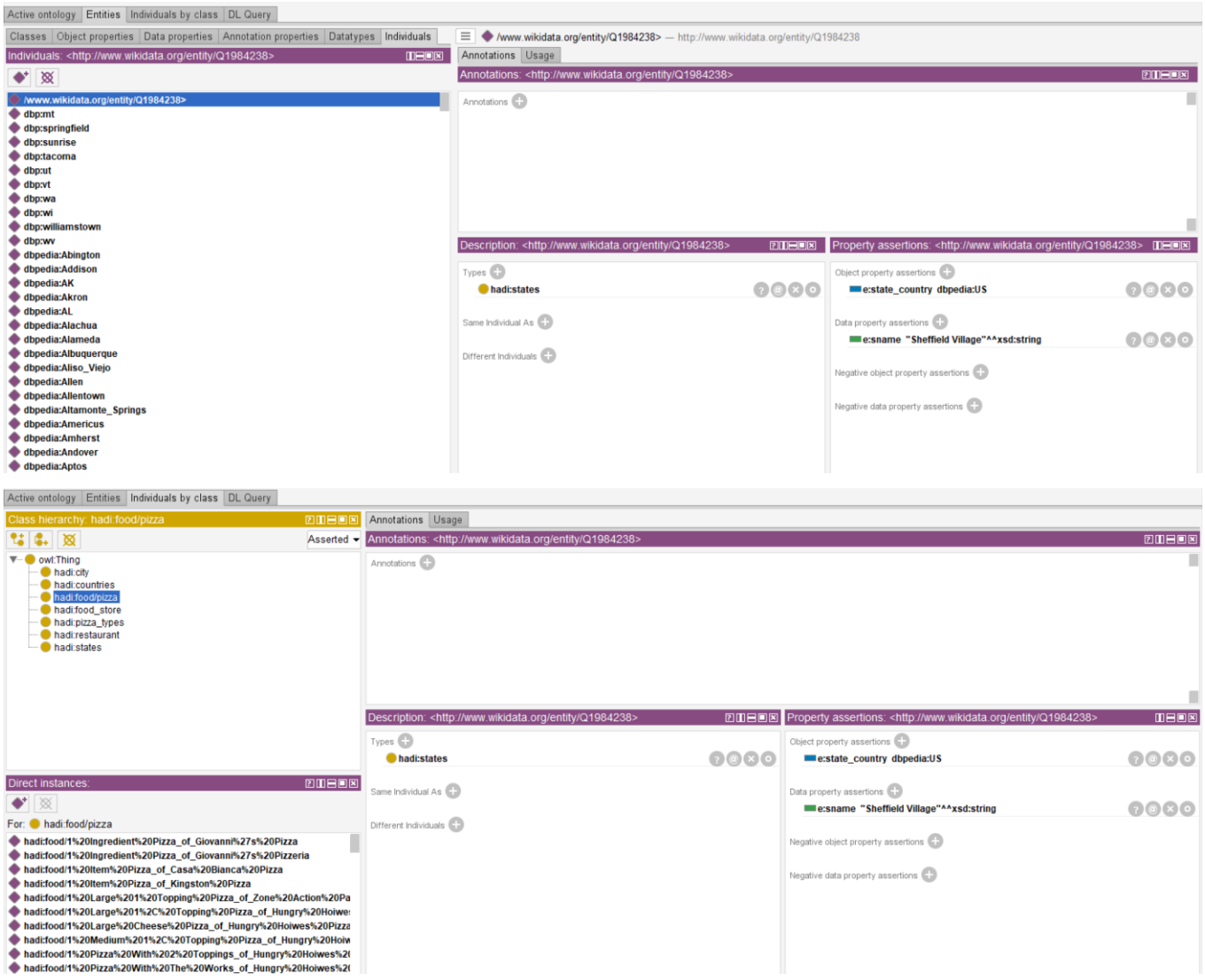
Final created knowledge graph:






Active ontology
Entities
Individuals by class
DL Query
Classes
Object properties
Data properties
Annotation properties
Datatypes
Individuals
e:menu — http://www.city.ac.uk/ds/inm713/hadi_ghasemi/relations/menu
Object property hierarchy: e:menu
Asserted
owl:topObjectProperty
e:category
e:city_state
e:menu
e:name_city
e:pizza_type
e:state_country
Annotations
Usage
Annotations: e:menu
Annotations
Characteristics: e
Description: e:menu
Functional
Inverse functional
Transitive
Symmetric
Asymmetric
Reflexive
Irreflexive
Equivalent To
SubProperty Of
Inverse Of
Domains (intersection)
Ranges (intersection)
Disjoint With
SuperProperty Of (Chain)


Active ontology
Entities
Individuals by class
DL Query
Classes
Object properties
Data properties
Annotation properties
Datatypes
Individuals
e:description — http://www.city.ac.uk/ds/inm713/hadi_ghasemi/relations/description
Data property hierarchy: e:description
Asserted
owl:topDataProperty
e:address
e:cname
e:description
e:pname
e:postcode
e:rname
e:sname
e:value
Annotations
Usage
Annotations: e:description
Annotations
Characteristics: e:d
Description: e:description
Functional
Equivalent To
SubProperty Of
Domains (intersection)
Ranges
Disjoint With


Active ontology
Entities
Individuals by class
DL Query
Classes
Object properties
Data properties
Annotation properties
Datatypes
Individuals
Individuals: <http://www.wikidata.org/entity/Q1984238>
/www.wikidata.org/entity/Q1984238> — http://www.wikidata.org/entity/Q1984238
Annotations
Usage
Annotations: <http://www.wikidata.org/entity/Q1984238>
/www.wikidata.org/entity/Q1984238>
dbp:mt
dbp:springfield
dbp:sunrise
dbp:tacoma
dbp:ut
dbp:vt
dbp:wa
dbp:wi
dbp:williamstown
dbp:wv
dbpedia:Abington
dbpedia:Addison
dbpedia:AK
dbpedia:Akron
dbpedia:AL
dbpedia:Alachua
dbpedia:Alameda
dbpedia:Albuquerque
dbpedia:Aliso_Viejo
dbpedia:Allen
dbpedia:Allentown
dbpedia:Altamonte_Springs
dbpedia:Americus
dbpedia:Amherst
dbpedia:Andover
dbpedia:Aptos
Description: <http://www.wikidata.org/entity/Q1984238>
Property assertions: <http://www.wikidata.org/entity/Q1984238>
Types
hadi:states
Same Individual As
Different Individuals
Object property assertions
e:state_country dbpedia:US
Data property assertions
e:sname "Sheffield Village"^^xsd:string
Negative object property assertions
Negative data property assertions
Active ontology
Entities
Individuals by class
DL Query
Class hierarchy: hadi:food/pizza
Asserted
owl:Thing
hadi:city
hadi:countries
hadi:food/pizza
hadi:food_store
hadi:pizza_types
hadi:restaurant
hadi:states
Annotations
Usage
Annotations: <http://www.wikidata.org/entity/Q1984238>
Description: <http://www.wikidata.org/entity/Q1984238>
Property assertions: <http://www.wikidata.org/entity/Q1984238>
Direct instances:
For: hadi:food/pizza
hadi:food/1%20Ingredient%20Pizza_of_Giovanni%27s%20Pizza
hadi:food/1%20Ingredient%20Pizza_of_Giovanni%27s%20Pizzeria
hadi:food/1%20Item%20Pizza_of_Casa%20Bianca%20Pizza
hadi:food/1%20Item%20Pizza_of_Kingston%20Pizza
hadi:food/1%20Large%201%20Topping%20Pizza_of_Zone%20Action%20Pa
hadi:food/1%20Large%201%2C%20Topping%20Pizza_of_Hungry%20Hoiwe
hadi:food/1%20Large%20Cheese%20Pizza_of_Hungry%20Hoiwes%20Pizza
hadi:food/1%20Medium%201%2C%20Topping%20Pizza_of_Hungry%20Hoiw
hadi:food/1%20Pizza%20With%202%20Toppings_of_Hungry%20Hoiwes%2
hadi:food/1%20Pizza%20With%20The%20Works_of_Hungry%20Hoiwes%2
Types
hadi:states
Same Individual As
Different Individuals
Object property assertions
e:state_country dbpedia:US
Data property assertions
e:sname "Sheffield Village"^^xsd:string
Negative object property assertions
Negative data property assertions