# -\*- coding: utf-8 -\*-

"""Coursework\_Solution.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1sHilBFadGfqMZzCZ9g0HPkkgqewKHBtQ

Adapted on May 2024

\*\*Author:\*\* Ali Momenzadeh kholejani & Pawel Staszynski

References:

\* Lab Solutions

\* https://en.wikipedia.org/wiki/Precision\_and\_recall

### Install Requirements

"""

!pip install rdflib

!pip install owlrl

!pip install owlready2

!pip install Levenshtein

"""### Import Packages"""

from owlready2 import \*

from rdflib import Graph

from rdflib import URIRef, BNode, Literal

from rdflib import Namespace

from rdflib.namespace import OWL, RDF, RDFS, FOAF, XSD

import pandas as pd

import owlrl

import Levenshtein as lev

"""### Mount Google Drive"""

from google.colab import drive

drive.mount('/content/drive')

"""### Data Exploration"""

base\_file\_url = '/content/drive/My Drive/KG\_Coursework'

file\_path = f'{base\_file\_url}/data/IN3067-INM713\_coursework\_data\_pizza\_500.csv'

df = pd.read\_csv(file\_path)

df.head()

df.shape

# checking the missing values

df.isna().sum()

"""### Data Preprocessing"""

# Removing unnecessary punctuation marks. Solution by: ChatGPT 3.5

import string

cols = ['name', 'address', 'city', 'country', 'state', 'categories', 'menu item']

translator = str.maketrans('', '', string.punctuation)

for col in cols:

df[col] = df[col].apply(lambda x: x.translate(translator)

.lower()

.replace('-', '\_')

.replace(',', '\_')

.replace(' ', '\_')

.replace('(', '')

.replace(')', '')

)

df.head()

"""### Encoding the Dataset

After we attempted to identify pizza types based on their ingredients using the find\_best\_matching\_pizza function, inconsistencies were observed in the ontology when utilizing the "ns.hasIngredient" property in "2.2 Tabular Data to Knowledge Graph (Task RDF)" section. Therefore, we assigned the generic type "Pizza" to instances where no matches were found in the pizza titles. The reasoner will be able to infer the type of pizza, rather than manually assigning scores or estimates based on ingredient composition.

"""

# Defining each type of pizza with their respective ingredients

pizza\_with\_ingredients = {

"bbq\_pizza": ["bbq\_sauce"],

"beans\_pizza":["beans"],

"chicken\_pizza": ["chicken"],

"feta\_pizza":["feta"],

"fruit\_pizza":["fig", "pineapple"],

"greek\_pizza":["feta", "black\_olives", "green\_olives", "spinach"],

"meat\_pizza":["bacon", "beef", "chicken", "chorizo", "ham", "meatballs", "mortadella",

"prosciutto", "salami", "sausage", "pepperoni"],

"mexican\_pizza":["beans", "hot\_sauce"],

"mushroom\_pizza":["mushroom"],

"hawaiian\_pizza":["tomato\_sauce", "marinara", "blue\_cheese", "cheddar", "feta", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta", "ham", "pineapple"],

"margherita\_pizza":["tomato\_sauce", "marinara", "basil", "mozzarella"],

"pizza\_napolitana":["tomato\_sauce", "marinara", "basil", "mozzarella",

"blue\_cheese", "cheddar", "feta", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta"],

"pizza\_marinara":["garlic", "mozzarella", "tomato\_sauce", "marinara"],

"pizza\_supreme":["onion", "green\_pepper", "jalapeno\_pepper", "red\_pepper",

"yellow\_pepper", "pepperoni", "sausage"],

"pizzaernesto": ["anchovies", "capers"],

"pineapple\_pizza":["pineapple"],

"pizza\_bianca":[],

"californian\_pizza":[],

"japanese\_pizza":["salmon", "tuna"],

"pizza\_romana":[],

"seafood\_pizza":["anchovies", "crab\_meat", "salmon", "scallop", "shrimp", "tuna"],

"pizza\_nutella":[],

"vegetarian\_pizza":["blue\_cheese", "cheddar", "feta", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta",

"bbq\_sauce", "olive\_oil", "tofu", "tomato\_sauce",

"marinara", "vegan\_cheese",

"artichoke", "broccoli", "capers", "eggplant",

"fig", "pineapple", "garlic", "basil", "oregano",

"rosemary", "mushroom", "black\_olives", "green\_olives",

"onion", "green\_pepper", "jalapeno\_pepper",

"red\_pepper", "yellow\_pepper", "carrot", "potato",

"sweet\_potato", "beans", "spinach", "tomato", "cherry\_tomato",

"plum\_tomato", "butternut\_squash", "pumpkin", "zucchini"],

"vegan\_pizza":["bbq\_sauce", "olive\_oil", "tofu", "tomato\_sauce",

"marinara", "vegan\_cheese",

"artichoke", "broccoli", "capers", "eggplant",

"fig", "pineapple", "garlic", "basil", "oregano",

"rosemary", "mushroom", "black\_olives", "green\_olives",

"onion", "green\_pepper", "jalapeno\_pepper",

"red\_pepper", "yellow\_pepper", "carrot", "potato",

"sweet\_potato", "beans", "spinach", "tomato", "cherry\_tomato",

"plum\_tomato", "butternut\_squash", "pumpkin", "zucchini"]

}

# Defining the ingredients that doesn't go together in a pizza

constraints = {

"mexican\_pizza":["beans", "hot\_sauce"], #meaning beans and hot\_sauce can't be both in a mexican\_pizza (Beans or some Hot Sauce)

"pizza\_napolitana": ["basil",

"blue\_cheese", "cheddar", "feta", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta",

"tomato\_sauce", "marinara"],

"pizza\_marinara": ["garlic", "mozzarella", "tomato\_sauce", "marinara"],

"pizzaernesto": ["anchovies", "capers"],

"japanese\_pizza":["salmon", "tuna"],

}

# Defining the ingredients that are not allowed in a pizza(if they are in a pizza, then the pizza is defintely of not that type)

incompatible\_ingredients = {

"pizza\_bianca": ["tomato\_sauce", "marinara"],

"vegetarian\_pizza": ["bacon", "beef", "chicken", "chorizo", "ham", "meatballs", "mortadella",

"prosciutto", "salami", "sausage", "pepperoni",

"anchovies", "crab\_meat", "salmon", "scallop", "shrimp", "tuna"], # Meat and Seafood are not allowed

"vegan\_pizza": ["bacon", "beef", "chicken", "chorizo", "ham", "meatballs", "mortadella",

"prosciutto", "salami", "sausage", "pepperoni",

"anchovies", "crab\_meat", "salmon", "scallop", "shrimp", "tuna",

"blue\_cheese", "cheddar", "feta", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta"] # Meat, Seafood, and Cheese are not allowed

}

def find\_best\_matching\_pizza(items, pizza\_types, pizza\_ingredients):

best\_match = None

max\_matching\_count = 0

for pizza in pizza\_types:

ingredients = pizza\_ingredients.get(pizza, [])

matching\_count = sum(item.lower() in ingredients for item in items)

if matching\_count > max\_matching\_count and check\_constraints(pizza, items, pizza\_ingredients):

best\_match = pizza

max\_matching\_count = matching\_count

return best\_match

def check\_constraints(pizza, items, pizza\_ingredients):

if pizza in constraints:

conflicting\_ingredients = constraints[pizza]

for ingredient in conflicting\_ingredients:

if ingredient.lower() in items and any(

conflicting\_ingredient.lower() in items

for conflicting\_ingredient in conflicting\_ingredients

if conflicting\_ingredient.lower() != ingredient.lower()):

# If any conflicting ingredient is present, return False

return False

if pizza in incompatible\_ingredients:

for incompatible\_pizza, incompatible\_ingredient in incompatible\_ingredients.items():

if pizza != incompatible\_pizza and any(item.lower() in incompatible\_ingredient for item in items):

# If an incompatible ingredient is present in another pizza type, return False

return False

return True

df['pizza\_type'] = '' # A new column to show the type of pizza

for idx, row in df.iterrows():

item\_list = row['menu item'].split() + str(row['item description']).split()

best\_matching\_pizza = None

# Iterate through the keys of pizza\_with\_ingredients

for pizza\_type, ingredients in pizza\_with\_ingredients.items():

# Check if any ingredient from the pizza type is in the item\_list

if pizza\_type in ' '.join(item\_list):

best\_matching\_pizza = pizza\_type

break

# If no exact match is found, use the function to find the best matching pizza

# if best\_matching\_pizza is None:

# best\_matching\_pizza = find\_best\_matching\_pizza(item\_list, pizza\_with\_ingredients.keys(), pizza\_with\_ingredients)

# If there is still no match, it belongs to the generic class of Pizza

if best\_matching\_pizza is None:

best\_matching\_pizza = "Pizza"

# Assign the pizza type to the "Pizza Type" column

df.at[idx, 'pizza\_type'] = best\_matching\_pizza.capitalize() # capitalize the first letter (we use this value later)

df.head(50)

# creating the instance name (we use this later)

df['instance\_name'] = (df['menu item'] + '\_at\_' + df['name'] + "\_" + df['city']).str.strip()

df.head(10)['instance\_name']

def capitalize\_second\_word(s):

words = s.split('\_')

if len(words) > 1:

words[1] = words[1].capitalize()

return ''.join(words)

df['pizza\_type'] = df['pizza\_type'].apply(capitalize\_second\_word)

df['pizza\_type'] = df['pizza\_type'].replace({'BbqPizza': 'BarbecuePizza'})

df.head(10)

# all the ingredients we have

all\_ingredients = [

"meat", "bacon", "beef", "chicken", "chorizo", "ham", "meatballs", "mortadella",

"prosciutto", "salami", "sausage", "pepperoni",

"sauce", "bbq\_sauce", "hot\_sauce", "pesto", "tomato\_sauce", "marinara",

"anchovies", "crab\_meat", "salmon", "scallop", "shrimp", "tuna",

"cheese", "blue\_cheese", "feta", "cheddar", "goat\_cheese", "gorgonzola",

"mozzarella", "parmesan", "provolone", "ricotta",

"olive\_oil", "tofu", "vegan\_cheese",

"vegetable", "artichoke", "broccoli", "capers", "eggplant",

"fruit", "fig", "pineapple","garlic",

"herbs", "basil", "oregano", "rosemary",

"mushroom", "olives", "black\_olives", "green\_olives",

"onion", "pepper", "green\_pepper", "jalapeno\_pepper", "red\_pepper", "yellow\_pepper",

"carrot", "potato", "sweet\_potato",

"beans", "spinach", "tomato", "cherry\_tomato", "plum\_tomato",

"butternut\_squash", "pumpkin", "zucchini"

]

# this function will iterate through the value of 'menu item' and 'item description' to capture the

# ingredients of each pizza and add it to 'matched\_ingredients' column. We will use this later.

def find\_matching\_ingredients(row):

matched\_ingredients = []

for item in all\_ingredients:

if item in row['menu item'].lower():

matched\_ingredients.append(item)

if isinstance(row['item description'], str):

description\_items = row['item description'].split(',')

for item in description\_items:

item = item.strip()

if ' ' in item:

item = '\_'.join(item.split())

for ingredient in all\_ingredients:

if ingredient in item.lower():

matched\_ingredients.append(ingredient)

return ', '.join(matched\_ingredients)

df['matched\_ingredients'] = df.apply(find\_matching\_ingredients, axis=1)

df['matched\_ingredients'] = df['matched\_ingredients'].astype(str)

df.head(30)

# we can also decide to delete the pizza that does not have any matched ingredients (but we did allow them)

# df = df[df['matched\_ingredients'].notna() & (df['matched\_ingredients'] != '')]

# df.head(30)

"""# 2.2 Tabular Data to Knowledge Graph (Task RDF)"""

def convert\_str\_to\_rdf\_friendly(input\_string):

words = input\_string.split('\_')

result = ''.join(word.capitalize() for word in words)

if result == 'BbqSauce':

return 'BarbecueSauce'

if result == 'Artichoke':

return 'Artichokes'

if result == 'Scallop':

return 'Scallops'

return result

# Empty graph

g = Graph()

# Note that this is the same namespace used in the ontology "pizza-restaurants-ontology.ttl"

namespace\_str = "http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#"

ns = Namespace(namespace\_str)

g.bind("", URIRef(namespace\_str))

# Prefixes for the serialization

g.bind("cw", ns)

# Adding creators annotation

g.add(

(URIRef(namespace\_str),

ns.Created\_by,

Literal('Ali Momenzadeh kholejani & Pawel Staszynski', datatype = RDFS.Literal)))

# Creating triples from ingredients

for item in all\_ingredients:

friendly\_name = convert\_str\_to\_rdf\_friendly(item)

ingredient = URIRef(namespace\_str+str(friendly\_name))

class\_uri = getattr(ns, friendly\_name)

g.add((ingredient, RDF.type, class\_uri))

# Creating triples from the df

for index, row in df.iterrows():

# name

restaurant = URIRef(namespace\_str+str(row['name']))

g.add((restaurant, RDF.type, ns.Restaurant))

g.add((restaurant , ns.restaurantName, Literal(row['name'], datatype = XSD.string)))

# restaurantName data property

g.add((restaurant, ns.restaurantName, Literal(row['name'], datatype = XSD.string)))

# pizza

pizza = URIRef(namespace\_str+str(row['instance\_name']))

class\_uri = getattr(ns, row['pizza\_type'])

g.add((pizza, RDF.type, class\_uri))

# itemName data property

g.add((pizza, ns.itemName, Literal(class\_uri, datatype = XSD.string)))

# servedIn, serves object properties

g.add((pizza, ns.servedIn, restaurant))

g.add((restaurant, ns.serves, pizza))

# item value

if row['item value'] == row['item value']:

item\_value = URIRef(namespace\_str+str(row['item value']))

g.add((item\_value, RDF.type, ns.ItemValue))

# amount data property

g.add((item\_value, ns.amount, Literal(row['item value'], datatype = XSD.double)))

if row['currency'] == row['currency']:

curreny = URIRef(namespace\_str+str(row['currency']))

g.add((curreny, RDF.type, ns.Currency))

#amountCurrency object property

g.add((item\_value, ns.amountCurrency, curreny))

#hasValue object property

g.add((pizza, ns.hasValue, item\_value))

# address, city, country, state

location = URIRef(namespace\_str+'Location')

address = URIRef(namespace\_str+row['address'])

city = URIRef(namespace\_str+row['city'])

country = URIRef(namespace\_str+row['country'])

state = URIRef(namespace\_str+row['state'])

g.add((location, RDF.type, ns.Location))

g.add((address, RDF.type, ns.Address))

# firstLineAddress data property

g.add((address, ns.firstLineAddress, Literal(row['address'], datatype = XSD.string)))

g.add((city, RDF.type, ns.City))

g.add((country, RDF.type, ns.Country))

g.add((state, RDF.type, ns.State))

#locatedIn, locatedInAddress, locatedInCity, locatedInCountry, locatedInState

g.add((state, ns.locatedInCountry, country))

g.add((city, ns.locatedInState, state))

g.add((address, ns.locatedInCity, city))

g.add((restaurant, ns.locatedInCity, city))

g.add((restaurant, ns.locatedInAddress, address))

# postCode

if row['postcode'] is not None:

# postCode data property

g.add((address, ns.postCode, Literal(row['postcode'], datatype = XSD.string)))

# ingredients

if row['matched\_ingredients'] is not None:

if isinstance(row['matched\_ingredients'], str):

items = row['matched\_ingredients'].split(',')

trimmed\_items = [item.strip() for item in items]

for item in trimmed\_items:

if len(item) > 0:

ingredient\_name = convert\_str\_to\_rdf\_friendly(item)

# hasIngredient object property

ingredient\_item = URIRef(namespace\_str+str(ingredient\_name))

g.add((pizza, ns.hasIngredient, ingredient\_item))

print("Triples count: (with no reasoning/parsing ontology): {}".format(len(g)))

"""### Subtask RDF.3"""

# import requests

# def get\_google\_kg\_uri(query, api\_key):

# """Retrieve the URI from Google Knowledge Graph."""

# url = "https://kgsearch.googleapis.com/v1/entities:search"

# params = {

# 'query': query,

# 'limit': 1,

# 'indent': True,

# 'key': api\_key

# }

# response = requests.get(url, params=params)

# json\_response = response.json()

# try:

# return json\_response['itemListElement'][0]['result']['@id']

# except (IndexError, KeyError):

# return None

# file\_path = '/IN3067-INM713\_coursework\_data\_pizza\_500(1).csv'

# df = pd.read\_csv(file\_path)

# print(df.head())

# print(df.columns)

# # API Key for Google Knowledge Graph

# api\_key = 'AIzaSyA6Bf9yuMCCPh7vpElzrfBvE2ENCVWr-84'

# kg\_source = {'city': 'google\_kg', 'country': 'google\_kg', 'state': 'google\_kg'}

# # Apply the function to each column based on the specified KG source

# for column, kg in kg\_source.items():

# if column in df.columns:

# unique\_values = df[column].unique()

# if kg == 'google\_kg':

# uri\_mapping = {value: get\_google\_kg\_uri(value, api\_key) for value in unique\_values}

# df[column] = df[column].map(uri\_mapping)

# else:

# print(f"Column {column} not found in DataFrame.")

# df.to\_csv('updated\_dataset\_with\_kg\_uris.csv', index=False)

# print(df)

# def get\_wikidata\_uri(query):

# """Retrieve the URI from Wikidata."""

# sparql = SPARQLWrapper("https://query.wikidata.org/sparql")

# sparql.setQuery(f"""

# SELECT ?item WHERE {{

# ?item ?label "{query}"@en.

# SERVICE wikibase:label {{ bd:serviceParam wikibase:language "[AUTO\_LANGUAGE],en". }}

# }}

# LIMIT 1

# """)

# sparql.setReturnFormat(JSON)

# results = sparql.query().convert()

# try:

# return results["results"]["bindings"][0]["item"]["value"]

# except (IndexError, KeyError):

# return None

"""### Save the results"""

print("Triples count (with no reasoning/parsing ontology): {}".format(len(g)))

g.serialize(f'{base\_file\_url}/output/pizza-restaurants-ontology-initial.ttl', format = 'ttl')

g.parse(f'{base\_file\_url}/pizza-restaurants-model-ontology/pizza-restaurants-ontology.ttl', format = 'ttl')

g.serialize(f'{base\_file\_url}/output/pizza-restaurants-ontology-extended.ttl', format = 'ttl')

print("Triples count (with parsing): {}".format(len(g)))

"""### Subtask RDF.4"""

def checkEntailment(g, triple):

qres = g.query(

"""ASK {""" + triple + """ }""")

#Single row with one boolean vale

for row in qres:

print("Does '" + triple + "' hold? " + str(row))

# RDFS reasoning via owlrl:

owlrl.DeductiveClosure(owlrl.OWLRL\_Semantics, axiomatic\_triples=True, datatype\_axioms=False).expand(g)

print("Number of triples after reasoning: {}".format(len(g)))

# some triples to check thier entailment:

# Each restaurant is located in a country

t1 = "?restaurant cw:locatedInCountry ?country ."

# Each restaurant serves a pizza with a specific pizza type

t2 = "?restaurant cw:serves ?pizza . ?pizza rdf:type ?pizza\_type ."

# Each pizza has an item value and currency

t3 = "?pizza cw:hasItemValue ?item\_value . ?item\_value cw:amount ?amount . ?item\_value cw:amountCurrency ?currency ."

checkEntailment(g, t1)

checkEntailment(g, t2)

checkEntailment(g, t3)

# save the graph:

g.serialize(f'{base\_file\_url}/output/pizza-restaurants-ontology-after-reasoning.ttl', format = 'ttl')

"""# 2.3 SPARQL and Reasoning (Task SPARQL)"""

import csv

def save\_to\_csv(qres, file\_path, column\_titles=None):

with open(file\_path, 'w', newline='') as csvfile:

writer = csv.writer(csvfile)

# if provided write column titles

if column\_titles:

writer.writerow(column\_titles)

# write rows

for row in qres:

writer.writerow(row)

csv\_files\_base\_url = f'{base\_file\_url}/output/queries\_output'

def queryGraph(query):

qres = g.query(query)

return qres

"""### Subtask SPARQL.1

This query returns the restaurants names and the pizzas they serve, only for the restaurants that thier name starts with "the".

"""

file\_path = f'{csv\_files\_base\_url}/sparql\_1.csv'

query\_text = """

PREFIX cw: <http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#>

SELECT ?restaurantName ?pizza

WHERE {

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

FILTER regex(?restaurantName, "^the")

}

"""

qres = queryGraph(query\_text)

column\_titles = ['Restaurant Name', 'Pizza']

save\_to\_csv(qres, file\_path, column\_titles)

"""### Subtask SPARQL.2

This query returns the restaurants names, pizzas, and thier ingredients only for the restaurants with names shorter than 10 charachters and pizzas with ingredients either "Olives" or "Cheese".

"""

file\_path = f'{csv\_files\_base\_url}/sparql\_2.csv'

query\_text = """

PREFIX cw: <http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#>

SELECT ?restaurantName ?pizza ?ingredient

WHERE {

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

?pizza cw:hasIngredient ?ingredient .

FILTER (STRLEN(?restaurantName) < 10

&& (?ingredient = cw:Olives || ?ingredient = cw:Cheese))

}

"""

qres = queryGraph(query\_text)

column\_titles = ['Restaurant Name', 'Pizza', 'Ingredient']

save\_to\_csv(qres, file\_path, column\_titles)

"""### Subtask SPARQL.3

This query returns restaurantName, pizzas, and their ingredient for pizzas containing either "Fig" or "Pineapple", together with pizzas containing "Olives".

"""

file\_path = f'{csv\_files\_base\_url}/sparql\_3.csv'

query\_text = """

PREFIX cw: <http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#>

SELECT ?restaurantName ?pizza ?ingredient

WHERE {

{

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

?pizza cw:hasIngredient ?ingredient .

FILTER (?ingredient = cw:Fig || ?ingredient = cw:Pineapple)

}

UNION

{

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

?pizza cw:hasIngredient ?ingredient .

FILTER (?ingredient = cw:Olives)

}

}

"""

qres = queryGraph(query\_text)

column\_titles = ['Restaurant Name', 'Pizza', 'Ingredient']

save\_to\_csv(qres, file\_path, column\_titles)

"""### Subtask SPARQL.4

This query returns the count of different pizzas served in each restaurant and filters out the restaurants that serve more that "8" types of pizza.

"""

file\_path = f'{csv\_files\_base\_url}/sparql\_4.csv'

query\_text = """

PREFIX cw: <http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#>

SELECT ?restaurantName (COUNT(?pizza) as ?numPizzas)

WHERE {

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

}

GROUP BY ?restaurantName

HAVING (COUNT(?pizza) > 8)

"""

qres = queryGraph(query\_text)

column\_titles = ['Restaurant Name', 'Number of Pizzas']

save\_to\_csv(qres, file\_path, column\_titles)

"""### Subtask SPARQL.5

This query returns restaurant names alongside counts of distinct pizzas and ingredients served by each of them. It then filters restaurants to get the ones which sells more than 5 pizzas and orders the results by pizza count in descending order and by ingredient count in ascending order.

"""

file\_path = f'{csv\_files\_base\_url}/sparql\_5.csv'

query\_text = """

PREFIX cw: <http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#>

SELECT ?restaurantName

(COUNT(DISTINCT ?pizza) AS ?numPizzas)

(COUNT(DISTINCT ?ingredient) AS ?numIngredients)

WHERE {

?pizza rdf:type cw:Pizza .

?pizza cw:servedIn ?restaurant .

?restaurant cw:restaurantName ?restaurantName .

?pizza cw:hasIngredient ?ingredient .

}

GROUP BY ?restaurantName

HAVING (COUNT(DISTINCT ?pizza) > 5)

ORDER BY DESC(?numPizzas) ASC(?numIngredients)

"""

qres = queryGraph(query\_text)

column\_titles = ['Restaurant Name', 'Number of Pizzas', 'Number of Ingredients']

save\_to\_csv(qres, file\_path, column\_titles)

"""# 2.4 Ontology Alignment (Task OA)

### Subtask OA.1

"""

pizza\_file\_url= f'{base\_file\_url}/pizza-ontology/pizza.owl'

csw\_file\_url = f'{base\_file\_url}/pizza-restaurants-model-ontology/pizza-restaurants-ontology.owl'

# reusing the functions from lab 8

def getClasses(onto):

return onto.classes()

def loadOntology(uri):

#Method from owlready

onto = get\_ontology(uri).load()

classes = []

print("Classes in Ontology: " + str(len(list(getClasses(onto)))))

for cls in getClasses(onto):

print("\t"+cls.name)

classes.append(cls.name)

return classes

def getObjectProperties(onto):

return onto.object\_properties()

def getObjectPropertiesList(uri):

#Method from owlready

onto = get\_ontology(uri).load()

props = []

print("Object Properties in Ontology: " + str(len(list(getObjectProperties(onto)))))

for prop in getObjectProperties(onto):

print("\t"+prop.name)

props.append(prop.name)

return props

pizza\_classes = loadOntology(pizza\_file\_url)

csw\_classes = loadOntology(csw\_file\_url)

g = Graph()

pizza\_namespace\_str = 'http://www.co-ode.org/ontologies/pizza/pizza.owl#'

csw\_namespace\_str = 'http://www.semanticweb.org/city/in3067-inm713/2024/restaurants#'

pizza\_ns = Namespace(pizza\_namespace\_str)

csw\_ns = Namespace(csw\_namespace\_str)

g.bind("pizza\_ns", pizza\_ns)

g.bind("csw\_ns", csw\_ns)

for cls in pizza\_classes:

if cls in csw\_classes:

pizza\_uri = URIRef(pizza\_namespace\_str+str(cls))

csw\_uri = URIRef(csw\_namespace\_str+str(cls))

# creating owl:equivalentClass triple

g.add((pizza\_uri, OWL.equivalentClass, csw\_uri))

elif ("Topping" in cls and cls == "PizzaTopping"):

pizza\_uri = URIRef(pizza\_namespace\_str+str(cls))

csw\_uri = URIRef(csw\_namespace\_str+str("PizzaTopping"))

g.add((pizza\_uri, OWL.equivalentClass, csw\_uri))

elif ("Topping" in cls and cls != "PizzaTopping"):

topping = cls.replace("Topping", "")

if topping in csw\_classes:

print(topping)

pizza\_uri = URIRef(pizza\_namespace\_str+str(cls))

csw\_uri = URIRef(csw\_namespace\_str+str(topping))

g.add((pizza\_uri, OWL.equivalentClass, csw\_uri))

else:

for cl in csw\_classes:

similarity = lev.jaro\_winkler(cls, cl)

if similarity > 0.85:

pizza\_uri = URIRef(pizza\_namespace\_str+str(cls))

csw\_uri = URIRef(csw\_namespace\_str+str(cl))

g.add((pizza\_uri, OWL.equivalentClass, csw\_uri))

len(g)

pizza\_props = getObjectPropertiesList(pizza\_file\_url)

csw\_props = getObjectPropertiesList(csw\_file\_url)

for prop in pizza\_props:

if prop in csw\_props:

pizza\_uri = URIRef(pizza\_namespace\_str+str(cls))

csw\_uri = URIRef(csw\_namespace\_str+str(cls))

# creating owl:equivalentProperty triple

g.add((pizza\_uri, OWL.equivalentProperty, csw\_uri))

else:

for pr in csw\_props:

similarity = lev.jaro\_winkler(prop, pr)

if similarity > 0.8:

pizza\_uri = URIRef(pizza\_namespace\_str+str(prop))

csw\_uri = URIRef(csw\_namespace\_str+str(pr))

g.add((pizza\_uri, OWL.equivalentProperty, csw\_uri))

len(g)

# save the graph:

system\_mapping\_file\_url = f'{base\_file\_url}/output/ontologies\_alignment\_task.ttl'

g.serialize(system\_mapping\_file\_url, format = 'ttl')

"""### Subtask OA.2"""

ref\_mapping\_file\_url= f'{base\_file\_url}/reference-mappings-pizza.ttl'

def compareWithReference(reference\_mappings\_file, system\_mappings\_file):

ref\_mappings = Graph()

ref\_mappings.parse(reference\_mappings\_file, format="ttl")

system\_mappings = Graph()

system\_mappings.parse(system\_mappings\_file, format="ttl")

# We calculate precision and recall via true positives, false positives and false negatives

# https://en.wikipedia.org/wiki/Precision\_and\_recall

tp = 0

fp = 0

fn = 0

for s, p, o in system\_mappings:

# Check if the triple exists in the reference mappings regardless of the order

if (s, p, o) in ref\_mappings or (o, p, s) in ref\_mappings:

tp += 1

else:

fp += 1

for s, p, o in ref\_mappings:

# Check if the triple exists in the system mappings regardless of the order

if (s, p, o) not in system\_mappings and (o, p, s) not in system\_mappings:

fn += 1

precision = tp / (tp + fp) if (tp + fp) != 0 else 0 # Avoid division by zero

recall = tp / (tp + fn) if (tp + fn) != 0 else 0 # Avoid division by zero

print("TP: " + str(tp))

print("FP: " + str(fp))

print("FN: " + str(fn))

print("Comparing '" + system\_mappings\_file + "' with '" + reference\_mappings\_file)

print("\tPrecision: " + str(precision))

print("\tRecall: " + str(recall))

compareWithReference(ref\_mapping\_file\_url, system\_mapping\_file\_url)

"""### Subtask OA.3"""

file\_path\_1 = f'{base\_file\_url}/pizza-restaurants-model-ontology/pizza-restaurants-ontology.ttl'

file\_path\_2 = f'{base\_file\_url}/pizza-ontology/pizza.ttl'

file\_path\_3 = f'{base\_file\_url}/output/ontologies\_alignment\_task.ttl'

file\_path\_4 = f'{base\_file\_url}/output/pizza-restaurants-ontology-extended.ttl'

g = Graph()

g.parse(file\_path\_1, format = 'ttl')

g.parse(file\_path\_2, format = 'ttl')

g.parse(file\_path\_3, format = 'ttl')

g.parse(file\_path\_4, format = 'ttl')

print("Number of triples before reasoning: {}".format(len(g)))

# RDFS reasoning via owlrl:

owlrl.DeductiveClosure(owlrl.OWLRL\_Semantics, axiomatic\_triples=True, datatype\_axioms=False).expand(g)

print("Number of triples after reasoning: {}".format(len(g)))

"""### Subtask OA.4

This query returns the pizza name, ingredients, and type of pizza for pizzas with the "VegetarianPizza" type.

"""

file\_path = f'{csv\_files\_base\_url}/alignment\_sparql.csv'

query\_text = """

PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>

SELECT ?pizza ?ingredient ?type

WHERE {

?pizza a pizza:Pizza ;

a ?type ;

pizza:hasIngredient ?ingredient .

FILTER (?type = pizza:VegetarianPizza)

}

"""

qres = g.query(query\_text)

column\_titles = ['Pizza', 'Ingredient', 'Type']

save\_to\_csv(qres, file\_path, column\_titles)

"""# 2.5 Ontology Embeddings (Task Vector)

Unfortunately, we couldn't finish this part due to time constraints. :(

### Subtask Vector.1

"""

"""### Subtask Vector.2"""