```
# -*- 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)
```

```python
"""# 2.5 Ontology Embeddings (Task Vector)


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


### Subtask Vector.1
"""




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