Rapport Web Datamining

Ontology

The ontology defined for the food delivery domain establishes a structured framework to represent entities, relationships, and attributes relevant to food delivery services. This ontology is encoded using RDF, RDFS, OWL, and other XML namespaces to provide a comprehensive and extensible model for describing the various aspects of the food delivery ecosystem, including participants, their interactions, and the products and services involved.

Core Classes and Hierarchies

- **Person**: Represents the generic concept of a person involved in the system. This class serves as a superclass for more specific roles.
    - **Customer**: A subclass of Person, representing individuals who order food.
    - **Delivery Person**: Another subclass of Person, denoting individuals responsible for delivering food orders.
- **Business**: Abstracts the concept of businesses operating within the food delivery domain.
    - **Restaurant**: Specializes the Business class to represent dining establishments.
    - **Delivery Service**: Another specialization of Business, focusing on entities that provide food delivery services.
- **Product**: Encompasses items that can be offered, ordered, and delivered within the system.
    - **Food Item**: A type of Product specifically representing different kinds of food.
    - **Offer**: Represents special deals or offerings that include Food Items, distinct from the FoodItem class to avoid confusion.

- **Order**: Captures details about orders placed by Customers, including what was ordered and the delivery specifics.
- **Location**: A class inherited from the VCard namespace to detail geographical locations, used for specifying delivery areas, restaurant locations, etc.
- **Preference**: Abstracts the concept of preferences that Customers can have.
    - **Cuisine Preference**: A specific type of Preference detailing the preferred cuisines of Customers.

Object Properties

The ontology defines various object properties to model the relationships between the different classes:

- `hasPreference`: Links Person entities to their Preferences.
- `offersFoodItem`: Associates Restaurants with the Food Items they offer.
- `deliveredBy` and `deliversFor`: Bidirectional properties linking Restaurants and Delivery Services.
- `includesFoodItem`: Connects Offers with the Food Items they include.

Datatype Properties

These properties assign specific data values to instances of classes, enabling detailed descriptions of entities:

- `hasCuisineType`: Specifies the cuisine type of a Food Item.
- `servesCuisine`: Indicates the type of cuisine a Restaurant serves.
- `age`: Records the age of a Delivery Person.

Restrictions and Class Specifications

The ontology uses OWL restrictions to enforce data integrity and define the semantics of the food delivery domain more precisely. For example:

- Restaurants must offer at least one Food Item and serve at least one type of cuisine.
- Customers must have at least one Preference.

- Orders must contain at least one Product.

These restrictions ensure that the ontology accurately models the real-world constraints and relationships within the food delivery ecosystem.

Python Code for importing data

This report provides an in-depth analysis of a Python script designed for generating RDF (Resource Description Framework) data for a hypothetical food delivery project. The script integrates various Python libraries and custom logic to create a comprehensive dataset that models restaurants, food items, customers, and delivery services within a semantic web framework.

We opted to use Python for this task due to its simplicity and efficiency. Querying using Protégé can often be complex and time-consuming. Python, being a high-level language with a clear syntax, makes it easier to create, manipulate and query RDF data. Furthermore, Python's extensive library support, such as RDFlib and pandas, greatly simplifies the process of working with RDF data.

Libraries and Imports

```
from rdflib import Graph, URIRef, Literal, Namespace, RDF, XSD
import pandas as pd
import random
import names
```

- **RDFlib**: A Python library for working with RDF, a standard model for data interchange on the web. The script uses RDFlib to create and manipulate the RDF graph that models the food delivery domain.
- **pandas**: A powerful data manipulation and analysis library for Python. It is used here to handle and process the dataset.
- **random**: This module implements pseudo-random number generators for various distributions and is used for selecting random elements within the script.

- **names**: An external library for generating random names, utilized for creating fictitious names for customers and delivery persons.

Namespace Definition

The script defines a custom namespace (`NS`) for the RDF data, indicating that all the resources created within this graph belong to a specific domain, in this case, the food delivery project's domain.

```
NS = Namespace("http://www.leonard-and-alex-project.com/foodDel
```

Data Model and Processing

Sanitization Function

```
def sanitize_for_uri(text):
    return ''.join(e for e in text if e.isalnum())
```

A utility function, `sanitize_for_uri`, ensures that text strings are clean and valid for URI (Uniform Resource Identifier) creation by removing any non-alphanumeric characters.

Loading and Sampling Data

```
df = pd.read_csv('zomato.csv', encoding='ISO-8859-1').sample(10
```

The script loads a dataset from a CSV file named `zomato.csv` using pandas, then randomly selects 100 samples for processing. This subset forms the basis for the RDF graph's data.

Food Items and Cuisines

```
food_items_by_cuisine = {
    "Italian": ["Pizza Margherita", "Spaghetti Carbonara", "Las
    .....
}
```

A predefined dictionary, `food_items_by_cuisine`, maps various cuisines to specific food items, demonstrating the script's ability to handle diverse food categories.

RDF Graph Creation

```
g = Graph()
```

An RDF graph (`g`) is instantiated, serving as the container for all the RDF data created by the script.

Adding Data to the RDF Graph

```
def add_food_items_and_offer(restaurant_uri, cuisine):
        if cuisine in food_items_by_cuisine:
                food_items = food_items_by_cuisine[cuisine]
                selected_items = random.sample(food_items, min(
                for food_item in selected_items:
                        food_item_uri = NS[sanitize_for_uri(food
                        g.add((food_item_uri, RDF.type, NS.Food
                        g.add((food_item_uri, NS['hasCuisineType
                        g.add((restaurant_uri, NS['offersFoodIte

                # Create an offer for the first two selected foo
                offer_uri = NS[sanitize_for_uri("Offer_for_" + 
                g.add((offer_uri, RDF.type, NS.Offer))
                offer_description = "Special offer for " + " and
                g.add((offer_uri, NS.description,
                             Literal(offer_description, datatype
                for food_item in selected_items[:2]:  # Apply of
                        food_item_uri = NS[sanitize_for_uri(food
                        g.add((offer_uri, NS.includesFoodItem, 
```

Restaurants, Food Items, and Offers

```
def add_customers_to_restaurant(restaurant_uri, cuisine):
        # Assume that the cuisine preference is based on the pri
        for _ in range(2):  # Add 2 customers for each restauran
                customer_name = names.get_full_name()
                customer_uri = NS[sanitize_for_uri(customer_name
                g.add((customer_uri, RDF.type, NS.Customer))
                g.add((customer_uri, NS.name, Literal(customer_n

                # Generate preference based on the restaurant's
                preference_name = "Preference_" + cuisine
                preference_uri = NS[sanitize_for_uri(preference_
                g.add((preference_uri, RDF.type, NS.CuisinePrefe
                g.add((preference_uri, NS.preferredCuisine, Lite

                # Link the customer to their preference
                g.add((customer_uri, NS.hasPreference, preferenc
```

The script includes functions for adding restaurants, food items, and special offers to the RDF graph. Each restaurant is linked to a set of food items based on its cuisine, and special offers are created for selected food items.

Customers and Preferences

```
def add_delivery_service_and_person(restaurant_uri, country_code
        country_code_str = str(country_code)
        # Check if a delivery service exists for this country
        if country_code_str not in delivery_companies_by_country
                # Create a new delivery service for this country
                delivery_service_name = "DeliveryService_" + cou
                delivery_service_uri = NS[sanitize_for_uri(deliv
                g.add((delivery_service_uri, RDF.type, NS.Delive
                g.add((delivery_service_uri, NS.name, Literal(de
                g.add((delivery_service_uri, NS.deliveryArea, Li
                delivery_companies_by_country[country_code_str]
        else:
                # Use the existing delivery service for this cou
                delivery_service_uri = delivery_companies_by_cou

        # Link the restaurant to the delivery service
        g.add((restaurant_uri, NS.deliveredBy, delivery_service_
        g.add((delivery_service_uri, NS.deliversFor, restaurant_

        # Generate a random name and age for the delivery persor
        random_name = names.get_full_name()
        random_age = random.randint(18, 60)  # Random age betwee

        delivery_person_name = "DeliveryPerson_" + sanitize_for_
        delivery_person_uri = NS[sanitize_for_uri(delivery_perso
        g.add((delivery_person_uri, RDF.type, NS.DeliveryPerson)
        g.add((delivery_person_uri, NS.name, Literal(random_name
        g.add((delivery_person_uri, NS.age, Literal(random_age,
        g.add((delivery_person_uri, NS.worksFor, delivery_servic
```

Another function adds customers with specific cuisine preferences to restaurants, modeling a realistic scenario where customers have preferred types of food.

Delivery Services and Personnel

```
def add_delivery_service_and_person(restaurant_uri, country_code
        country_code_str = str(country_code)
        # Check if a delivery service exists for this country
        if country_code_str not in delivery_companies_by_country
                # Create a new delivery service for this country
                delivery_service_name = "DeliveryService_" + cou
                delivery_service_uri = NS[sanitize_for_uri(deliv
                g.add((delivery_service_uri, RDF.type, NS.Delive
                g.add((delivery_service_uri, NS.name, Literal(de
                g.add((delivery_service_uri, NS.deliveryArea, Li
                delivery_companies_by_country[country_code_str]
        else:
                # Use the existing delivery service for this cou
                delivery_service_uri = delivery_companies_by_cou

        # Link the restaurant to the delivery service
        g.add((restaurant_uri, NS.deliveredBy, delivery_service_
        g.add((delivery_service_uri, NS.deliversFor, restaurant_

        # Generate a random name and age for the delivery person
        random_name = names.get_full_name()
        random_age = random.randint(18, 60)  # Random age betwee

        delivery_person_name = "DeliveryPerson_" + sanitize_for_
        delivery_person_uri = NS[sanitize_for_uri(delivery_perso
        g.add((delivery_person_uri, RDF.type, NS.DeliveryPerson)
        g.add((delivery_person_uri, NS.name, Literal(random_name
        g.add((delivery_person_uri, NS.age, Literal(random_age,
        g.add((delivery_person_uri, NS.worksFor, delivery_servic
```

The script also accounts for the delivery aspect by adding delivery services and delivery personnel. Each restaurant is associated with a delivery service, which in turn is linked to a delivery person.

Dual-role Entities

```
def create_dual_entities(entity_name, address, cuisine, delivery
        entity_uri = NS[sanitize_for_uri(entity_name)]

        # Add the entity as both a Restaurant and a DeliveryServ
        g.add((entity_uri, RDF.type, NS.Restaurant))
        g.add((entity_uri, RDF.type, NS.DeliveryService))

        # Adding common attributes
        g.add((entity_uri, NS.name, Literal(entity_name)))
        g.add((entity_uri, NS.locatedIn, Literal(address)))
        g.add((entity_uri, NS.servesCuisine, Literal(cuisine)))
        g.add((entity_uri, NS.deliveryArea, Literal(delivery_are
```

In a more advanced use case, the script demonstrates creating entities that serve dual roles, such as a restaurant that also acts as a delivery service. This showcases the flexibility of RDF in modeling complex real-world relationships.

Serialization and Output

```
output_path = 'FoodDelivery.owl'
g.serialize(destination=output_path, format='pretty-xml')
print(f"RDF data generated and saved to: {output_path}")
```

Finally, the script serializes the RDF graph to an RDF/XML format, saving the output to a file named `FoodDelivery.owl`. This step makes the data ready for use in other applications, further analysis, or sharing.

Queries

SPARQL Queries for Food Delivery Data Analysis

This section explores the application of SPARQL queries to interrogate and extract information from the RDF graph created by the food delivery ontology and the corresponding Python script. SPARQL, a powerful query language for RDF, enables sophisticated querying capabilities to retrieve, manipulate, and infer data from RDF graphs.

SPARQL Query Execution Setup

Before diving into specific queries, the setup involves loading the RDF graph with both the ontology structure (`ontology.owl`) and the instance data (`FoodDelivery.owl`), as shown below:

```
from rdflib import Graph, RDF, Namespace
from rdflib.plugins.sparql import prepareQuery
```

```
NS = Namespace("http://www.leonard-and-alex-project.com/foodDel
g = Graph()
g.parse("ontology.owl", format="xml")
g.parse("FoodDelivery.owl", format="xml")
```

Listing Customers and Delivery Persons

```
PREFIX ns: <http://www.leonard-and-alex-project.com/foodDelivery
SELECT ?name ?type WHERE {
  {
    ?person a ns:Customer.
    BIND("Customer" AS ?type)
  } UNION {
    ?person a ns:DeliveryPerson.
    BIND("Delivery Person" AS ?type)
  }
  ?person ns:name ?name.
}
```

Complex Queries

```
PREFIX ns: <http://www.leonard-and-alex-project.com/foodDelivery
CONSTRUCT {
  ?restaurant ns:offersCuisine ?cuisine .
}
WHERE {
  ?restaurant rdf:type ns:Restaurant ;
              ns:servesCuisine ?cuisine .
}
```

```
PREFIX ns: <http://www.leonard-and-alex-project.com/foodDelivery
SELECT ?restaurantName ?deliveryService
WHERE {
  ?restaurant rdf:type ns:Restaurant ;
              ns:servesCuisine ?cuisine ;
              ns:name ?restaurantName .
  FILTER ( ?cuisine = "Bakery" )
  OPTIONAL { ?restaurant ns:deliveredBy ?deliveryService . }
}
```

The complex queries provided are written in SPARQL, a powerful query language for RDF data. Here's a breakdown of each one:

1. **Constructing a graph of restaurants and cuisines they offer**
   This query constructs a new graph where each restaurant is linked to the cuisines it serves. The
   `CONSTRUCT` clause specifies the structure of the triples in the new graph. The `WHERE` clause specifies the pattern to match in the existing data. In this case, the pattern is "restaurants serving a certain cuisine". This query is useful in creating a simplified view of the data focusing on restaurants and the cuisines they serve.

2. **Retrieving restaurants serving Bakery cuisine and their associated delivery services**
   This query retrieves the names of all restaurants serving 'Bakery' cuisine and the delivery services associated with them. The
   `SELECT` clause specifies which variables to return (restaurant names and delivery services). The `WHERE` clause specifies the pattern to match: "restaurants serving Bakery cuisine". The `FILTER` clause restricts the results to only those where the cuisine is 'Bakery'. The `OPTIONAL` clause includes the associated delivery service, if it exists. If a restaurant does not have an associated delivery service, the query still returns the restaurant's name but leaves the delivery service field blank. This query helps in finding all bakeries and their respective delivery services in the dataset.

Conclusion

In conclusion, the food delivery ontology provides an effective framework for representing and organizing a complex food delivery ecosystem. The use of Python in conjunction with RDF and SPARQL allows for efficient data manipulation and querying, allowing users to generate insights and understand the relationships within the system. It demonstrates the power of semantic web technologies in managing complex data structures and relationships. However, it's important to remember that the ontology and data model should be continually refined and expanded to better represent the evolving real-world domain.