Decoding IPL 2022

Comprehensive Data on Matches and Players using Ontology

Extraction of the Data

For this Project, I have taken two different datasets namely IPL Data1.csv and IPL Data2.csv.

- Dataset 1 was taken from https://www.kaggle.com/datasets/patrickb1912/ipl-complete-dataset-20082020 it has the data from the year 2008 – 2023, I extracted only 2022 data to be concise.
- Dataset 2 was taken from https://www.crictracker.com/ipl-winners-and-runners-list/, In that I took IPL Winners List with Captain, Man of the Match, and Player of the Series table data.
- The first IPL Data1.owl was created using the first dataset and the second IPL Data2.owl was created using the second dataset.
- After this, both datasets are merged into a single ontology named MergedOntology.owl.

Creating the First Ontology

Creating Classes

Using the First dataset, the classes are created, those classes are City, Match, People, Results, Stadium_venue and Teams.

1. Class Match

- Class Match has subclasses ranges from Match_1, Match_2, Match_3 till Match_Eliminator_(72)
- These classes are disjoint with the other classes.
- The subclasses which are present under the Class Match are also disjoint with the other subclasses.

2. Class City

- Class City is disjoint with the other classes (i.e) Match, People, Results, Stadium_venue and Teams.

3. Class People

- Class People has two subclasses namely Players and Umpires.
- Class People is disjoint with the other classes.
- The subclasses Players and Umpires are disjoint with each other.

4. Class Results

- Class Results has subclasses namely Game_winner, Toss_decision, Toss_winner and Won_by.
- The subclass Won_by has its own subclass namely Runs and Wickets.
- Class Results is disjoint with the other classes.
- The subclasses are disjoint with the other subclasses.

5. Class Stadium_venue

- Class Stadium_venue has subclasses namely Brabourne_Stadium_Mumbai, Dr_DY_Patil_Sports_Academy_Mumbai, Eden_Gardens_Kolkata, Maharashtra_Cricket_Association_Stadium_Pune, Narendra_Modi_Stadium_Ahmedabad and Wankhede_Stadium_Mumbai.
- Class Stadium is disjoint with the other classes.
- The subclasses are all disjoint with each other.

6. Class Teams

- Class Teams has subclasses namely Chennai_Super_Kings, Delhi_Capitals, Gujarat_Titans, Kolkata_Knight_Riders, Lucknow_Super_Giants, Mumbai_Indians, Punjab_kings, Rajasthan_Royals, Royal_Challengers_Bangalore and Sunrisers_Hyderabad.
- Class Teams is disjoint with the other classes.
- The subclasses are all disjoint with each other.

Creating Object Properties

The following are the Object properties created in the first ontology

1) GAME_WINNER

Domains: Match

Ranges: Teams

2) HAS_PLAYER_NAME

Domains: Match

Ranges: Players

3) HAS_STADIUM

Domains: Match

Ranges:Stadium_venue

4) HAS_TOSS_DECISION

Domains: Umpires, Match

Ranges: Toss_decision

5) PLAYER_OF_THE_MATCH

Domains: Match

Ranges: Players

6) PLAYING_TEAM_1

Domains: Match

Ranges: Teams

7) PLAYING_TEAM_2

Domains: Match

Ranges: Teams

8) RUNS_WON

Domains: Match

Ranges: Runs

9) TEAM_1_PLAYERS

Domains: Match

Ranges: Players

10) TEAM_2_PLAYERS

Domains: Match

Ranges: Players

11) UMPIRE_1

Domains: Match

Ranges: Umpires

12) UMPIRE_2

Domains: Match

Ranges: Umpires

13) WICKETS_WON

Domains: Match

Ranges: Wickets

Creating Data Properties

The following are the Object properties created in the first ontology

1) HAS_MATCH_NUMBER

Domains: Runs, Wickets

Ranges: xsd:integer

2) HAS_VENUE

Domains: City

Ranges: xsd:string

3) IS_PLAYER_OF

Domains: Players

Ranges: xsd:string

Challenges Faced during this Project:

- A significant difficulty arose in determining whether a particular value should be represented as a class or an individual instance. This decision was crucial as it impacted the creation and mapping of object and data properties accordingly.
- Specifying the linkage between object and data properties within parent classes required the use of specific keywords, which added a layer of complexity to the process.
- The Protégé ontology editor did not provide a direct option to save the ontology in the widely-used .owl file format, which would have been a more convenient choice.

How did I solved the Challenge:

To determine the initial requirements of the project and understand the process of constructing the ontology, the coursework example provided in QM+ and the pizza ontology from the lab work served as the primary references. These examples were the only resources utilized for designing the ontology, as no other references were given or found online. Although the ontology was saved in the Turtle syntax, a widely used format for representing RDF data, it is ultimately represented as an OWL file, which is the standard format for ontologies.

Code for creating object property:

```
!pip install rdflib
import csv
from rdflib import Graph, Literal, Namespace, RDF, URIRef, OWL, XSD

ont =
Namespace("http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2
024/3/untitled-ontology-14/")
xsd = Namespace("http://www.w3.org/2001/XMLSchema#")

g = Graph()
g.parse("1st_ontology.ttl", format="turtle")

has_venue = ont.HAS_VENUE
g.add((has_venue, RDF.type, OWL.ObjectProperty))

isPlayerOf = ont.IS_PLAYER_OF
g.add((isPlayerOf, RDF.type, OWL.DatatypeProperty))

hasMatchNumber = ont.HAS_MATCH_NUMBER
g.add((hasMatchNumber, RDF.type, OWL.ObjectProperty))

def format_uri_component(name):
    return name.strip().replace(" ", "_").replace(",", "").replace(".",
"").replace("\\", "")

with open("IPL Dataset 1.csv", "r", newline="", encoding='utf-8') as
csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        umpire1_uri = URIRef(ont[format_uri_component(row["Umpire1"])])
        g.add((umpire1_uri, RDF.type, ont.Umpires))
        umpire2_uri = URIRef(ont[format_uri_component(row["Umpire2"])])
        g.add((umpire2_uri, RDF.type, ont.Umpires))

        city_uri = URIRef(ont[format_uri_component(row["City"])])
        g.add((city_uri, RDF.type, ont.City))

        venue_name = Literal(row["Venue"], datatype=XSD.string)
        g.add((city_uri, has_venue, venue_name))

        if row["WonBy"] == "Wickets":
            wickets_uri =
URIRef(ont[format_uri_component(row["Margin"])])
            match_number = Literal(row["MatchNumber"],
datatype=XSD.string)
            g.add((wickets_uri, hasMatchNumber, match_number))
            g.add((wickets_uri, RDF.type, ont.Wickets))
        elif row["WonBy"] == "Runs":
            runs_uri = URIRef(ont[format_uri_component(row["Margin"])])
            g.add((runs_uri, RDF.type, ont.Runs))
            g.add((runs_uri, hasMatchNumber, match_number))

        toss_uri =
URIRef(ont[format_uri_component(row["TossDecision"])])
        g.add((toss_uri, RDF.type, ont.Toss_decision))

        g.add((umpire1_uri, ont.HAS_TOSS_DECISION, toss_uri))
        g.add((umpire2_uri, ont.HAS_TOSS_DECISION, toss_uri))

        for team_key, player_key in [("Team1", "Team1Players"),
("Team2", "Team2Players")]:
            team_uri = Literal(row[team_key], datatype=XSD.string)
            player_list = eval(row[player_key])
            for player in player_list:
                player_uri = URIRef(ont[format_uri_component(player)])
                g.add((player_uri, RDF.type, ont.Players))
                g.add((player_uri, isPlayerOf, team_uri))

g.serialize(destination="1st_ontology_populated.ttl", format="turtle")
```

Explanation of the Code Snippet:

- The code begins by installing the required rdflib library and importing necessary modules. It then defines custom namespaces for ontology terms and data types. Next, it initializes an RDF graph and parses an existing Turtle file containing RDF data.
- The ontology properties, such as has_venue, isPlayerOf, and hasMatchNumber, are declared with their respective types (ObjectProperty or DatatypeProperty) to specify how they relate to ontology classes. A helper function is defined to format string values into valid URI components.
- The main functionality involves processing a CSV file containing IPL (Indian Premier League) match data. For each row in the CSV, the code creates and connects RDF resources and literals based on the data. It creates URIs for umpires, cities, and associate’s venues with cities. Match outcomes (won by wickets or runs) are represented using appropriate URIs and literals, with relationships defined using relevant properties. Toss decisions related to umpires are also recorded.
- Additionally, for each team mentioned in a row, the code processes the list of players, creates URIs for each player, and links them to their respective teams using the isPlayerOf property.
- Finally, the populated RDF graph is serialized and saved as a Turtle file named "1st_ontology_populated.ttl," containing all the RDF data generated from the CSV file.

Code:

```
!pip install rdflib
import pandas as pd
from rdflib import Graph, Namespace, RDF, OWL, URIRef, Literal, BNode,
RDFS

ontology_path = '1st_ontology_populated.ttl'

g = Graph()
g.parse(ontology_path, format="turtle")

ont =
Namespace("http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2
024/3/untitled-ontology-14/")
g.bind("ont", ont)

wickets_won_uri = URIRef(ont["WICKETS_WON"])
runs_won_uri = URIRef(ont["RUNS_WON"])

dataset_path = 'IPL Dataset 1.csv'
data = pd.read_csv(dataset_path)

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))
    if row['WonBy'] == 'Wickets':
        value_uri = URIRef(ont[str(row['Margin'])])
        restriction = BNode()
        g.add((restriction, RDF.type, OWL.Restriction))
        g.add((restriction, OWL.onProperty, wickets_won_uri))
        g.add((restriction, OWL.hasValue, value_uri))
    elif row['WonBy'] == 'Runs':
        value_uri = URIRef(ont[str(row['Margin'])])
        restriction = BNode()
        g.add((restriction, RDF.type, OWL.Restriction))
        g.add((restriction, OWL.onProperty, runs_won_uri))
        g.add((restriction, OWL.hasValue, value_uri))

    g.add((match_uri, RDFS.subClassOf, restriction))
```

Explanation of the code:

- The code begins by importing the necessary modules, including pandas for handling CSV data and various rdflib classes for RDF operations. It also installs the rdflib library if it's not already installed.
- Next, it loads an existing RDF graph from a Turtle file (1st_ontology_populated.ttl), which contains some predefined ontology data. The code then defines and binds a namespace (ont) that is used to reference the ontology elements.
- URIs for properties WICKETS_WON and RUNS_WON are defined, which are used to describe how matches are won in the dataset.
- The match data from the "IPL Dataset 1.csv" file is loaded into a pandas DataFrame. This data is then iterated row-by-row to update the RDF graph with match-specific details.
- For each match (row in the DataFrame), the code constructs a URI for the match based on its number (MatchNumber), ensuring any spaces are replaced with underscores to maintain URI integrity. It checks if the match URI already exists as a class in the graph. If not, it adds it as a new class (OWL.Class).
- Depending on the column WonBy (whether the match is won by "Wickets" or "Runs"), the code performs the following:

1. Creates a URI for the Margin (the number of wickets or runs by which the match was won).

2. Instantiates a blank node (BNode) which serves as an anonymous RDF resource.

3. Uses this blank node to create an OWL restriction, defining that the match (as a class) has a certain property (WICKETS_WON or RUNS_WON) with a specific value (value_uri, which represents the margin of victory).

4. Adds the created restriction as a subclass of the match class, establishing that the match class is constrained by the specified win condition (either by wickets or runs).

Finally, there is no serialization command in this snippet, so the updated graph exists only in memory during the script's execution.

Code for creating individuals:

```python
player_of_the_match_uri = URIRef(ont["PLAYER_OF_THE_MATCH"])
umpire_1_uri = URIRef(ont["UMPIRE_1"])
umpire_2_uri = URIRef(ont["UMPIRE_2"])
team_1_players_uri = URIRef(ont["TEAM_1_PLAYERS"])
team_2_players_uri = URIRef(ont["TEAM_2_PLAYERS"])

def process_player_list(player_list_str):
    player_names = player_list_str.strip("[]").replace("'",
"").split(", ")
    return [URIRef(ont[name.replace(' ', '_')]) for name in
player_names if name]

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))

    if pd.notna(row['Team1Players']):
        team1_players_uris = process_player_list(row['Team1Players'])
        for player_uri in team1_players_uris:
            restriction = BNode()
            g.add((restriction, RDF.type, OWL.Restriction))
            g.add((restriction, OWL.onProperty, team_1_players_uri))
            g.add((restriction, OWL.hasValue, player_uri))
            g.add((match_uri, RDFS.subClassOf, restriction))

    if pd.notna(row['Team2Players']):
        team2_players_uris = process_player_list(row['Team2Players'])
        for player_uri in team2_players_uris:
            restriction = BNode()
            g.add((restriction, RDF.type, OWL.Restriction))
            g.add((restriction, OWL.onProperty, team_2_players_uri))
            g.add((restriction, OWL.hasValue, player_uri))
            g.add((match_uri, RDFS.subClassOf, restriction))
```

Explanation of the Code:

- The code begins by defining URIs for various entities in the ontology, such as PLAYER_OF_THE_MATCH, UMPIRE_1, UMPIRE_2, TEAM_1_PLAYERS, and TEAM_2_PLAYERS. These URIs are used to reference specific properties that describe roles and teams within the matches.
- A function called process_player_list is defined to handle the processing of player lists. It takes a string representing a list of player names, formats it correctly by removing unnecessary characters (like brackets and quotes), and splits it into individual names. Each name is then formatted (spaces replaced with underscores) and converted into a URI that references a player within the ontology.
- The script iterates over each match in the DataFrame data, extracted from a CSV file. For each match, it performs the following operations:

1. Constructs a URI for the match based on the match number, following a consistent format (Match_{MatchNumber}).

2. Checks if the match URI already exists as a class in the RDF graph. If not, it adds it as a new OWL.Class.

- For each match, the script checks if the player lists for Team 1 and Team 2 (Team1Players and Team2Players) are not empty. If they are not empty, it calls the process_player_list function to convert the string of player names into URIs.
- For each player URI obtained, the script creates an RDF BNode (a blank node that functions as an anonymous resource) to serve as an OWL restriction. It then adds RDF statements to the graph, specifying the following:

1. The restriction is an OWL.Restriction, which relates to a property (team_1_players_uri or team_2_players_uri).

2. The match has a value (the player URI) for that property, effectively stating that the player is part of the team for that match.

3. The restriction is added as a subclass of the match, indicating that the match includes these player-team relationships as conditions or characteristics.

- By performing these operations, the script enriches the RDF graph with match-specific details, including the teams and players involved in each match.

Code:

```
playing_team_1_uri = URIRef(ont["PLAYING_TEAM_1"])
playing_team_2_uri = URIRef(ont["PLAYING_TEAM_2"])

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))

    if pd.notna(row['Team1']):
        team1_name = row['Team1'].replace(' ', '_')
        team1_class_uri = URIRef(ont[team1_name])
        if (team1_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team1_class_uri, RDF.type, OWL.Class))
        restriction1 = BNode()
        g.add((restriction1, RDF.type, OWL.Restriction))
        g.add((restriction1, OWL.onProperty, playing_team_1_uri))
        g.add((restriction1, OWL.allValuesFrom, team1_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction1))

    if pd.notna(row['Team2']):
        team2_name = row['Team2'].replace(' ', '_')
        team2_class_uri = URIRef(ont[team2_name])
        if (team2_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team2_class_uri, RDF.type, OWL.Class))
        restriction2 = BNode()
        g.add((restriction2, RDF.type, OWL.Restriction))
        g.add((restriction2, OWL.onProperty, playing_team_2_uri))
        g.add((restriction2, OWL.allValuesFrom, team2_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction2))
```

Expalantion of the Code:

- The code begins by defining URIs for the properties PLAYING_TEAM_1 and PLAYING_TEAM_2. These properties will be used to establish links between teams and their respective matches in the RDF graph.
- The script then iterates over rows in the DataFrame data, where each row contains information about a specific IPL (Indian Premier League) match. For each match, a unique URI (match_uri) is constructed based on the match number. This URI serves as the identifier for the match within the RDF graph.
- Next, the script checks if the match_uri already exists as an OWL.Class in the RDF graph. If it does not exist, the code adds it, effectively declaring each match as a class within the ontology.
- For each team involved in the match (Team1 and Team2), the following operations are performed:

1. The team name is converted into a URI-friendly format by replacing spaces with underscores, and a URI (team1_class_uri or team2_class_uri) is generated.

2. The script verifies if the team's URI already exists as a class in the RDF graph. If it does not exist, the code adds it, ensuring that each team is represented as a distinct class.

3. An RDF BNode (blank node) is instantiated to serve as an OWL restriction. This is a way to express constraints or conditions in OWL ontologies.

4. The type of the BNode is set to OWL.Restriction, and it is assigned a property (playing_team_1_uri or playing_team_2_uri) to denote that this match involves a specific team.

5. The restriction is configured to involve all values from the team's URI (team1_class_uri or team2_class_uri), meaning that the match class is constrained by the participation of this specific team.

6. The restriction is added as a subclass to the match URI, effectively linking the match to the team under the specified conditions.

- By performing these operations, the script enriches the RDF graph with information about the teams involved in each match and establishes the relationships between matches and teams using OWL restrictions.

Code:

```
wickets_won_uri = URIRef(ont["WICKETS_WON"])
runs_won_uri = URIRef(ont["RUNS_WON"])
player_of_the_match_uri = URIRef(ont["PLAYER_OF_THE_MATCH"])
umpire_1_uri = URIRef(ont["UMPIRE_1"])
umpire_2_uri = URIRef(ont["UMPIRE_2"])

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))

    if row['WonBy'] == 'Wickets':
        value_uri = URIRef(ont[str(row['Margin'])])
        restriction = BNode()
        g.add((restriction, RDF.type, OWL.Restriction))
        g.add((restriction, OWL.onProperty, wickets_won_uri))
        g.add((restriction, OWL.hasValue, value_uri))
        g.add((match_uri, RDFS.subClassOf, restriction))
    elif row['WonBy'] == 'Runs':
        value_uri = URIRef(ont[str(row['Margin'])])
        restriction = BNode()
        g.add((restriction, RDF.type, OWL.Restriction))
        g.add((restriction, OWL.onProperty, runs_won_uri))
        g.add((restriction, OWL.hasValue, value_uri))
        g.add((match_uri, RDFS.subClassOf, restriction))

    if pd.notna(row['Player_of_Match']):
        player_name = row['Player_of_Match'].replace(' ', '_')
        player_uri = URIRef(ont[player_name])
        player_restriction = BNode()
        g.add((player_restriction, RDF.type, OWL.Restriction))
        g.add((player_restriction, OWL.onProperty,
player_of_the_match_uri))
        g.add((player_restriction, OWL.hasValue, player_uri))
        g.add((match_uri, RDFS.subClassOf, player_restriction))

    if pd.notna(row['Umpire1']):
        umpire1_name = row['Umpire1'].replace(' ', '_')
        umpire1_uri = URIRef(ont[umpire1_name])
        umpire1_restriction = BNode()
        g.add((umpire1_restriction, RDF.type, OWL.Restriction))
        g.add((umpire1_restriction, OWL.onProperty, umpire_1_uri))
        g.add((umpire1_restriction, OWL.hasValue, umpire1_uri))
        g.add((match_uri, RDFS.subClassOf, umpire1_restriction))

    if pd.notna(row['Umpire2']):
        umpire2_name = row['Umpire2'].replace(' ', '_')
        umpire2_uri = URIRef(ont[umpire2_name])
        umpire2_restriction = BNode()
        g.add((umpire2_restriction, RDF.type, OWL.Restriction))
        g.add((umpire2_restriction, OWL.onProperty, umpire_2_uri))
        g.add((umpire2_restriction, OWL.hasValue, umpire2_uri))
        g.add((match_uri, RDFS.subClassOf, umpire2_restriction))
```

Expalanation of the Code:

- The code begins by defining URIs for various properties related to cricket matches, such as wickets_won_uri and runs_won_uri (representing win conditions), player_of_the_match_uri (for the player of the match), and umpire_1_uri and umpire_2_uri (for the first and second umpires).
- The script iterates through each row in a DataFrame data, which holds information about various cricket matches. For each match, a unique match_uri is constructed based on the match number, ensuring that each match can be distinctly identified in the RDF graph.
- Next, the code checks if the match_uri already exists as an OWL.Class in the graph. If it does not exist, it is added as a new class, categorizing each match as a distinct entity within the ontology.
- Depending on whether the match is won by wickets or runs, the script creates an OWL restriction (using a blank node, BNode) for the specific match outcome. A URI is created for the margin of victory (number of wickets or runs), and the restriction specifies that the match has the property wickets_won_uri or runs_won_uri with the specific value indicating the margin of victory. This restriction is then added as a subclass to the match_uri, implying that the match is constrained to having this specific outcome property.
- If a player of the match is specified, the player's name is converted into a URI, and a similar OWL restriction is created. This restriction connects the match to having a specific player as the player of the match via the player_of_the_match_uri and is added as a subclass of the match_uri.
- For umpires listed in the row, their names are processed into URIs, and OWL restrictions are set up for each. For each umpire, a restriction is created indicating that the match has this specific umpire assigned to it, associated via

umpire_1_uri or umpire_2_uri. These restrictions help define the roles of the umpires within each match and are added as subclasses to the match_uri.

- By performing these operations, the script enriches the RDF graph with details about match outcomes, players of the match, and assigned umpires, establishing relationships between matches and these entities using OWL restrictions.

Code:

```python
playing_team_1_uri = URIRef(ont["PLAYING_TEAM_1"])
playing_team_2_uri = URIRef(ont["PLAYING_TEAM_2"])
game_winner_uri = URIRef(ont["GAME_WINNER"])

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))

    if pd.notna(row['Team1']):
        team1_name = row['Team1'].replace(' ', '_')
        team1_class_uri = URIRef(ont[team1_name])
        if (team1_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team1_class_uri, RDF.type, OWL.Class))
        restriction1 = BNode()
        g.add((restriction1, RDF.type, OWL.Restriction))
        g.add((restriction1, OWL.onProperty, playing_team_1_uri))
        g.add((restriction1, OWL.allValuesFrom, team1_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction1))

    if pd.notna(row['Team2']):
        team2_name = row['Team2'].replace(' ', '_')
        team2_class_uri = URIRef(ont[team2_name])
        if (team2_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team2_class_uri, RDF.type, OWL.Class))
        restriction2 = BNode()
        g.add((restriction2, RDF.type, OWL.Restriction))
        g.add((restriction2, OWL.onProperty, playing_team_2_uri))
        g.add((restriction2, OWL.allValuesFrom, team2_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction2))

    if pd.notna(row['WinningTeam']):
        winner_name = row['WinningTeam'].replace(' ', '_')
        winner_class_uri = URIRef(ont[winner_name])
        if (winner_class_uri, RDF.type, OWL.Class) not in g:
            g.add((winner_class_uri, RDF.type, OWL.Class))
        restriction_winner = BNode()
        g.add((restriction_winner, RDF.type, OWL.Restriction))
        g.add((restriction_winner, OWL.onProperty, game_winner_uri))
        g.add((restriction_winner, OWL.allValuesFrom,
winner_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction_winner))
```

Explanation of the Code:

- The code begins by defining URIs for properties related to teams and match outcomes, such as playing_team_1_uri, playing_team_2_uri, and game_winner_uri. These properties represent the teams playing as Team 1 and Team 2 in a match, and the winning team, respectively.
- The script iterates over each row in the DataFrame data, which presumably contains information about a cricket match, including team names and the match number. For each match, a unique match_uri is constructed using the match number, acting as a unique identifier for the match within the RDF graph.
- The code then checks if the match_uri already exists as an OWL.Class in the graph. If it does not exist, it is added as a new class, representing the match as an ontology class within the graph.
- For processing the team data, the following steps are performed for both Team 1 and Team 2:

1. The team's name is converted into a standardized format (team1_name and team2_name) and used to create a corresponding team_class_uri.

2. If the team_class_uri does not already exist as a class in the RDF graph, it is added.

3. An OWL restriction is created (using a blank node or BNode), specifying that for the match represented by match_uri, the property (playing_team_1_uri or playing_team_2_uri) must have all values from (OWL.allValuesFrom) the respective team_class_uri. This effectively sets up a semantic condition that the team plays as Team 1 or Team 2 in this match.

4. This restriction is added as a subclass to the match, linking the team's participation in the match within the ontology.

- For handling the winning team, if a winning team is specified (WinningTeam), its name is processed into a URI (winner_class_uri). The script checks if this winner_class_uri exists as a class in the RDF graph. If it does not exist, it is added.
- An OWL restriction is then set up for the winner, stating that the match (match_uri) has the winning team identified by game_winner_uri with all

values from the winner_class_uri. This restriction is added as a subclass of the match, reinforcing the match's result within the ontology.
- By performing these operations, the script enriches the RDF graph with information about the teams participating in each match, their roles (Team 1 or Team 2), and the winning team, establishing relationships between these entities using OWL restrictions.

Code:

```
playing_team_1_uri = URIRef(ont["PLAYING_TEAM_1"])
playing_team_2_uri = URIRef(ont["PLAYING_TEAM_2"])
game_winner_uri = URIRef(ont["GAME_WINNER"])
has_stadium_uri = URIRef(ont["HAS_STADIUM"])

for index, row in data.iterrows():
    match_id = f"Match_{row['MatchNumber']}"
    match_uri = URIRef(ont[match_id].replace(" ", "_"))

    if (match_uri, RDF.type, OWL.Class) not in g:
        g.add((match_uri, RDF.type, OWL.Class))

    if pd.notna(row['Venue']):
        stadium_name = row['Venue'].replace(' ', '_').replace(',',
'').replace('.', '')  # Normalize the stadium name
        stadium_class_uri = URIRef(ont[stadium_name])
        if (stadium_class_uri, RDF.type, OWL.Class) not in g:
            g.add((stadium_class_uri, RDF.type, OWL.Class))
        restriction_stadium = BNode()
        g.add((restriction_stadium, RDF.type, OWL.Restriction))
        g.add((restriction_stadium, OWL.onProperty, has_stadium_uri))
        g.add((restriction_stadium, OWL.allValuesFrom,
stadium_class_uri))
        g.add((match_uri, RDFS.subClassOf, restriction_stadium))
```

Explanation of the Code:

- The code begins by defining URIs for properties related to teams, winners, and venues. While the URIs for playing_team_1_uri, playing_team_2_uri, and game_winner_uri (representing teams and winners) are defined, they are not directly used in the shown code snippet. However, the has_stadium_uri property is utilized to link matches to their respective venues (stadiums).
- The script iterates over each row in the DataFrame data, which includes information about various cricket matches, such as the match number and the venue. For each match, a unique match_uri is generated based on the match number, ensuring that each match is uniquely identifiable in the RDF graph.

- Next, the code checks if the match_uri already exists as an OWL.Class in the RDF graph. If it does not exist, it is added as a new class, making each match a distinct entity within the ontology.
- For processing venue data, the following steps are performed:

1. If a venue is specified for the match (Venue column), the venue name is normalized by removing spaces, commas, and periods to create a standardized URI (stadium_class_uri).

2. The script checks if this venue URI (stadium_class_uri) exists as a class in the RDF graph. If it does not exist, it is added, ensuring that each venue is represented as a distinct class.

3. An OWL restriction is then created using a blank node (BNode), which specifies that the match denoted by match_uri is associated with the venue through the has_stadium_uri property.

4. The restriction states that all values for this property must be from the stadium_class_uri, effectively linking the match to this particular stadium.

5. This restriction is added as a subclass of the match URI, integrating the venue information into the match's ontology class representation.

- By performing these operations, the script enriches the RDF graph with information about the venues where matches took place, establishing the relationships between matches and their respective stadiums using OWL restrictions.

Code:

```
team_1_players_uri = URIRef(ont["HAS_PLAYER_NAME"])
team_2_players_uri = URIRef(ont["HAS_PLAYER_NAME"])

for index, row in data.iterrows():
    if pd.notna(row['Team1']) and pd.notna(row['Team1Players']):
        team1_name = row['Team1'].replace(' ', '_')
        team1_class_uri = URIRef(ont[team1_name])

        if (team1_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team1_class_uri, RDF.type, OWL.Class))

        players = row['Team1Players'].strip("[]").replace("'",
"").split(", ")
        for player in players:
            player_uri = URIRef(ont[player.replace(' ', '_')])
            restriction = BNode()
            g.add((restriction, RDF.type, OWL.Restriction))
            g.add((restriction, OWL.onProperty, team_1_players_uri))
            g.add((restriction, OWL.hasValue, player_uri))
            g.add((team1_class_uri, RDFS.subClassOf, restriction))

    if pd.notna(row['Team2']) and pd.notna(row['Team2Players']):
        team2_name = row['Team2'].replace(' ', '_')
        team2_class_uri = URIRef(ont[team2_name])

        if (team2_class_uri, RDF.type, OWL.Class) not in g:
            g.add((team2_class_uri, RDF.type, OWL.Class))

        players = row['Team2Players'].strip("[]").replace("'",
"").split(", ")
        for player in players:
            player_uri = URIRef(ont[player.replace(' ', '_')])
            restriction = BNode()
            g.add((restriction, RDF.type, OWL.Restriction))
            g.add((restriction, OWL.onProperty, team_2_players_uri))
            g.add((restriction, OWL.hasValue, player_uri))
            g.add((team2_class_uri, RDFS.subClassOf, restriction))


output_ttl_path = '1st_ontology_output.ttl'
g.serialize(destination=output_ttl_path, format='turtle')
```

Explanation of the Code:

- The code begins by defining URIs for the properties team_1_players_uri and team_2_players_uri, both pointing to the same ontology property HAS_PLAYER_NAME. This property is intended to link teams to their respective players, with the identical URIs suggesting that both team properties are managed under the same predicate, implying a shared nature of the player association across teams.
- The script iterates through each row of the DataFrame data, which presumably contains information about various cricket matches, including team names and player lists for two teams.
- For each match, if the row has non-null entries for Team1 and Team1Players, the following steps are performed:

1. The team name (team1_name) is formatted by replacing spaces with underscores to create a standardized URI (team1_class_uri), uniquely identifying the team in the RDF graph.

2. The script checks if this URI exists as an OWL.Class in the graph and adds it if absent, designating each team as a distinct entity.

3. Player names from the Team1Players string are extracted and processed. Each player's name is formatted similarly (spaces replaced with underscores) to create individual player URIs.

4. For each player, a new RDF BNode (blank node) is created to represent an OWL restriction, indicating that the team has a particular player (specified by player_uri). This restriction uses the HAS_PLAYER_NAME property to associate the player with the team.

5. Each player-specific restriction is added as a subclass to the team's class URI, effectively stating that being associated with these players is a defining characteristic of the team.

- A similar process is followed for Team2 and Team2Players, where team and player URIs are created and checked, and OWL restrictions are generated for each player and linked to the team's class.
- Finally, after processing all the data, the completed RDF graph is serialized to a Turtle (TTL) file at a specified path (vidhyaa.ttl). The Turtle file format is a popular way to represent RDF data in a readable text format, allowing for easy storage and sharing of the enriched ontology data.

Extraction of data using SPARQL query:

```
!pip install rdflib
import rdflib
from rdflib.plugins.sparql import prepareQuery
from IPython.display import display, Image


team_name_query = """
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?team_name
WHERE {

<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/A_Badoni> rdf:type ma:Players ;
           ma:IS_PLAYER_OF ?team_name .
}
"""

venue_name_query = """
```

```
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?venue_name
WHERE {

<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/Ahmedabad> rdf:type ma:City ;
           ma:HAS_VENUE ?venue_name .
}
"""

g = rdflib.Graph()
g.parse("vidhyaa.ttl", "turtle")
results = g.query(team_name_query)
results_2 = g.query(venue_name_query)

for row in results:
    team_name = str(row.team_name).split('/')[-1]
    print("Name of the team for A_Badoni:", team_name)


for row in results_2:
    venue_name = str(row.venue_name).split('/')[-1]
    print("venue name for ahmedabad:", venue_name)
```

Detailed Explanation of the code:

- It installs the rdflib library.
- It imports the required modules and classes from rdflib, including rdflib itself, prepareQuery from rdflib.plugins.sparql, and display and Image from IPython.display.
- It defines two SPARQL queries as Python strings: team_name_query and venue_name_query. The former retrieves the team name associated with a player named "A_Badoni" in the ontology, while the latter fetches the venue name associated with the city "Ahmedabad".
- It creates an instance of rdflib.Graph, representing the RDF graph.
- It parses the "vidhyaa.ttl" file in Turtle format and loads its contents into the RDF graph instance.
- It executes the defined SPARQL queries against the loaded RDF graph, storing the results in variables results and results_2, respectively.
- It iterates over the results of each query:

- For team_name_query, it extracts the team name from the URI by splitting the URI on the '/' character and taking the last part of the split string. It then prints the team name associated with the player "A_Badoni".

- For venue_name_query, it follows a similar process, extracting the venue name from the URI and printing the venue name associated with the city "Ahmedabad".

Code for merging two ontologies:

```python
import csv
from rdflib import Graph, Literal, Namespace, RDF, URIRef, OWL
from rdflib.namespace import XSD
import ast

ont_18 =
Namespace("http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2
024/3/untitled-ontology-18/")
ont_14 =
Namespace("http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2
024/3/untitled-ontology-14/")
xsd = Namespace("http://www.w3.org/2001/XMLSchema#")

g = Graph()
g.parse("Merged_ontology.ttl", format="turtle")

isWinnerYear = ont_14.isWinnerYear
g.add((isWinnerYear, RDF.type, OWL.ObjectProperty))

hasWinnerCaptain = ont_18.hasWinnerCaptain
g.add((hasWinnerCaptain, RDF.type, OWL.ObjectProperty))

hasWinnerTeam = ont_18.hasWinnerTeam
g.add((hasWinnerTeam, RDF.type, OWL.ObjectProperty))

def get_winning_year(team_name):
  result = []
  with open("IPL.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
      if row["Winner"] == team_name:
        result.append(row["Year"])
  return result

def get_winning_captain(team_name):
  result = ""
  with open("IPL.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
      if row["Winner"] == team_name:
        result = row["Captain"]
  return result

def get_winning_team(year):
  result = ""
  with open("IPL.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
      if row["Year"] == year:
        result = row["Winner"]
  return result

# Read data from stadiums CSV file
with open("IPL Dataset 1.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
      team1_players = ast.literal_eval(row["Team1Players"])
      team1 = row["Team1"]
      winning_years = get_winning_year(team1)
      winning_captain = get_winning_captain(team1)
      for player in team1_players:
        player_uri = URIRef(ont_14[player.replace(" ", "_")])
        g.add((player_uri, hasWinnerCaptain, Literal(winning_captain)))
        for year in winning_years:
          g.add((player_uri, isWinnerYear, URIRef(ont_18[str(year)])))

      team2_players = ast.literal_eval(row["Team2Players"])
      team2 = row["Team2"]
      winning_years = get_winning_year(team2)
      winning_captain = get_winning_captain(team2)
      for player in team2_players:
        player_uri = URIRef(ont_14[player.replace(" ", "_")])
        g.add((player_uri, hasWinnerCaptain, Literal(winning_captain)))
        for year in winning_years:
          g.add((player_uri, isWinnerYear, URIRef(ont_18[str(year)])))

      seasons_uri = URIRef(ont_18[str(row["Season"])])
      g.add((seasons_uri, hasWinnerTeam,
Literal(str(get_winning_team(row["Season"])))))

g.serialize(destination="merged_ontology_populated.ttl",
format="turtle")
```

Explanation of the Code:

- The code defines three helper functions to extract specific information from a CSV file named "IPL.csv". These functions retrieve the winning years for a given team, the winning captain for a given team, and the winning team for a given year, respectively.
- Next, the code opens another CSV file named "IPL Dataset 1.csv" and reads its contents row by row. For each row, it performs the following operations:

1. Extracts the list of players for Team 1 and Team 2 from the CSV data.

2. Calls the helper functions to obtain the winning years and captains for both teams.

3. For each player in Team 1 and Team 2:

  - Creates a URI for the player by replacing spaces with underscores in their name.

  - Adds triples to the graph associating the player with their winning captain and winning year(s) using the defined properties.

4. Creates a URI for the season and adds a triple associating the season with the winning team using the hasWinnerTeam property.

- Finally, the updated RDF graph is serialized and saved to a new Turtle file named "merged_ontology_populated.ttl".

Code for populating merged dataset:

```
import rdflib

# Assuming the necessary imports and installations are handled outside
this script
g = rdflib.Graph()
try:
    g.parse("Vidhya_merged_ontology_populated.ttl", format="turtle")
    print("File parsed successfully")
except Exception as e:
    print("Failed to parse file:", e)

year_query = """
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-18/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?year
WHERE {
```

```
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-18/2022> rdf:type ma:Match_year ;
           ma:hasWinnerTeam ?year .
}
"""

try:
    results = g.query(year_query)
    if results:
        print("Query executed successfully, results obtained")
    else:
        print("Query executed but no results")
except Exception as e:
    print("Failed to execute query:", e)

for row in results:
    year_name = str(row.year).split('/')[-1]
    print("Winner team name for 2022:", year_name)
```

Explanation for the code:

- The code begins by creating an instance of the rdflib.Graph class, which represents the RDF graph. It then attempts to parse the "Vidhya_merged_ontology_populated.ttl" file in the Turtle format. If the file is parsed successfully, it prints a success message; otherwise, it catches the exception and prints an error message.
- Next, the code defines a SPARQL query as a string variable named year_query. This query selects the winning team name for the year 2022. The query uses PREFIX statements to define namespace prefixes and a pattern that specifies that the resource representing the year 2022 is of type Match_year and has the property hasWinnerTeam with the value bound to the variable ?year.
- The code then attempts to execute the SPARQL query against the parsed RDF graph using the g.query() method. If the query execution is successful and results are obtained, it prints a success message. If the query execution is successful but no results are found, it prints a message indicating no results. If an exception occurs during query execution, it catches the exception and prints an error message.
- Finally, the code iterates over the results obtained from the query. For each row in the results, it extracts the value of the ?year variable, which represents the winning team name. The winning team name is obtained by splitting the URI string on the '/' character and taking the last part of the split string. The

code then prints the message "Winner team name for 2022:" followed by the extracted winning team name.

- This code parses an RDF Turtle file, executes a SPARQL query to retrieve the winning team name for the year 2022, and prints the result if the query is successful and returns a result. It handles exceptions related to file parsing and query execution, providing appropriate error messages.

Code for Unmerged SPARQL Query:

```
!pip install rdflib
import rdflib
from rdflib.plugins.sparql import prepareQuery
from IPython.display import display, Image


team_name_query = """
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?team_name
WHERE {

<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/A_Badoni> rdf:type ma:Players ;
           ma:IS_PLAYER_OF ?team_name .
}
"""

venue_name_query = """
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?venue_name
WHERE {

<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-14/Ahmedabad> rdf:type ma:City ;
           ma:HAS_VENUE ?venue_name .
}
"""

g = rdflib.Graph()
g.parse("1st_ontology_output.ttl", "turtle")
```

```
results = g.query(team_name_query)
results_2 = g.query(venue_name_query)

for row in results:
    team_name = str(row.team_name).split('/')[-1]
    print("Name of the team for A_Badoni:", team_name)


for row in results_2:
    venue_name = str(row.venue_name).split('/')[-1]
    print("venue name for ahmedabad:", venue_name)
```

SPARQL for Merged ontology:

```
!pip install rdflib
import rdflib

# Assuming the necessary imports and installations are handled outside
this script
g = rdflib.Graph()
try:
    g.parse("Merged_ontology_populated.ttl", format="turtle")
    print("File parsed successfully")
except Exception as e:
    print("Failed to parse file:", e)

year_query = """
PREFIX ma:
<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-18/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?year
WHERE {

<http://www.semanticweb.org/vidhyaashreerajakuma/ontologies/2024/3/unti
tled-ontology-18/2022> rdf:type ma:Match_year ;
           ma:hasWinnerTeam ?year .
}
"""

try:
    results = g.query(year_query)
    if results:
        print("Query executed successfully, results obtained")
    else:
        print("Query executed but no results")
except Exception as e:
    print("Failed to execute query:", e)

for row in results:
    year_name = str(row.year).split('/')[-1]
    print("Winner team name for 2022:", year_name)
```

How to run the code:

1. For the FirstOntology.ipynb file, give 1st_ontology_unpopulated.ttl, 1st_ontology_populated and IPL Dataset 1.csv file and run it.

2. For the SecondOntology.ipynb file, give 2nd_ontology_unpopulated.ttl, 2nd_ontology_populated and IPL Dataset 2.csv file and run it.

3. For the MergedOntology.ipynb file, give IPL Dataset 1.csv, IPL Dataset 2.csv file, Merged_ontology_unpopulated.

4. For SPARQL, give 1st_ontology_output.ttl and run it.

5. For SPARQL_FOR_MERGED.ipynb, give Merged_ontology_populated.ttl and run it.

SWRL has been implemented for both individual and Merged Dataset, that is included in the zip file