CSV – OWL Bridge

Project- IP

Topic- CSV – OWL Bridge

Professor- Raghava Mutharaju

Members- Anand (2020280), Rohit (2020538)

1. Build Tool

Working:

This Python script is designed to read a list of required packages from a 'requirements.txt' file and install them using the pip package manager if they are not already installed. Here's a breakdown of how the code works:

install(package) function: This function takes a package name as an argument and installs it using the subprocess.check_call method. It calls the pip install command using the current Python interpreter (sys.executable).

main() function: Checks if the 'requirements.txt' file exists in the current directory. If the file is not found, it prints an error message and exits. Reads the list of required packages from 'requirements.txt'. Iterates through each package in the list:

Strips any leading or trailing whitespaces. Tries to import the package. If successful, it means the package is already installed, and a message is printed to indicate this. If importing the package raises an ImportError, it means the package is not installed, and the script attempts to install it using the install() function.

__name__ == "__main__": block: This block ensures that the main() function is only executed if the script is run directly (not imported as a module).

Algorithms Being Used:

The script uses the pip install command to install packages, which is a common way to manage Python packages. It uses the subprocess module to execute shell commands from within the

script. The try-except block is used to check whether a package is already installed. If an ImportError occurs during the import attempt, it means the package is not installed, and the script proceeds to install it.

The script is designed to be run from the command line, and it's particularly useful when setting up a Python environment for a project by automating the installation of required packages listed in a 'requirements.txt' file.

2. OWL – OWL Comparison

Working: This code performs an evaluation of two OWL (Web Ontology Language) files using various similarity and comparison techniques. Let's break down the code and discuss the algorithms being used:

1. XML Structural Comparison (deep_compare_elements and structural_similarity functions):
deep_compare_elements: A recursive function that compares two XML elements and their descendants for deep equality.

structural_similarity: Uses deep_compare_elements to compare the root elements of two XML files.

2. Advanced Semantic Comparison (wu_palmer_similarity and advanced_semantic_comparison functions):

wu_palmer_similarity: Calculates the Wu-Palmer similarity between two words using WordNet synsets.

advanced_semantic_comparison: Compares individuals in two OWL ontologies using Wu-Palmer similarity. Iterates over individuals in both ontologies. Calculates the similarity between the names of individuals using wu_palmer_similarity. Computes the average semantic score.

3. Advanced Isomeric Comparison (isomeric_comparison_advanced function): Uses Owlready2 to reason over ontologies. Converts individuals and their relationships into graphs using NetworkX. Checks if the graphs are isomorphic (structurally equivalent).

4. Evaluation Matrix (create_detailed_evaluation_matrix function): Creates a NumPy array with labels and scores for structural, semantic, and isomeric comparisons.

5. Plotting (plot_evaluation_matrix function): Uses Matplotlib to create a bar chart of the evaluation matrix. Bar colors represent different criteria, and the heights represent the scores. Text annotations on the bars show the percentage (scaled) or raw score.

6. File Paths and Comparison Execution: Specifies file paths for two OWL files: file1 and file2.

Calls structural_similarity, advanced_semantic_comparison, and isomeric_comparison_advanced to obtain respective scores. Loads the OWL ontologies using Owlready2 (onto1 and onto2).

7. Output: Prints the average semantic score, numeric score for structural similarity, and isomeric score.

8. Visualization: Calls create_detailed_evaluation_matrix to form an evaluation matrix. Calls plot_evaluation_matrix to visualize the matrix as a bar chart.

In summary, the code compares two OWL files based on their structural, semantic, and isomeric aspects. It utilizes XML comparison for structural analysis, WordNet and Wu-Palmer similarity for semantic analysis, and NetworkX for isomeric (graph-based) comparison. The evaluation results are presented in a bar chart for easy interpretation.

Algorithms Being Used:

1. **XML Structural Comparison (deep_compare_elements and structural_similarity functions):**

deep_compare_elements function:

Algorithm: Recursively compares XML elements and their attributes for deep equality. It iterates through child elements and compares each one.

How it works: It checks if the tag, text, attributes, and the number of child elements are the same. Recursively compares child elements.

structural_similarity function:

Algorithm: Parses two XML files using xml.etree.ElementTree. Calls deep_compare_elements to compare the root elements of the XML trees.

How it works: Parses XML files into ElementTree objects. Compares the root elements using deep_compare_elements.

2. **Advanced Semantic Comparison (wu_palmer_similarity and advanced_semantic_comparison functions):**

wu_palmer_similarity function:

Algorithm: Uses WordNet and the Wu-Palmer similarity metric. Finds WordNet synsets for input words and calculates the similarity.

How it works: Uses NLTK's lesk to find WordNet synsets for input words. Calculates Wu-Palmer similarity using the found synsets.

advanced_semantic_comparison function:

Algorithm: Compares individuals in two ontologies based on their names. Uses wu_palmer_similarity for semantic comparison. Calculates the average semantic score.

How it works: Iterates through individuals in both ontologies. Converts individual names to lowercase and calculates semantic similarity using wu_palmer_similarity. Computes the average semantic score.

3. **Advanced Isomeric Comparison (isomeric_comparison_advanced function):**

isomeric_comparison_advanced function:

Algorithm: Uses Owlready2 for ontology reasoning and NetworkX for graph representation.

Checks if the graphs of individuals in two ontologies are isomorphic.

How it works: Applies the Pellet reasoner using sync_reasoner_pellet. Converts individuals and their relationships into NetworkX graphs. Checks if the graphs are isomorphic using nx.is_isomorphic.

4. **Evaluation Matrix Creation (create_detailed_evaluation_matrix function):**

create_detailed_evaluation_matrix function:

Algorithm: Creates a NumPy array with labels and scores for structural, semantic, and isomeric comparisons.

How it works: Creates a 2D NumPy array with labels and corresponding scores.

5. **Plotting (plot_evaluation_matrix function):**

plot_evaluation_matrix function:

Algorithm: Uses Matplotlib to create a bar chart of the evaluation matrix. Adds text annotations to display scores on the bars.

How it works: Creates a bar chart using plt.bar. Adds text annotations with percentage or raw score values. Sets labels and title for the plot.

6. **File Paths and Comparison Execution:**

Specifies file paths for two OWL files: file1 and file2. Calls the relevant functions to perform structural, semantic, and isomeric comparisons.

7. **Output:**

Prints the average semantic score, numeric score for structural similarity, and isomeric score.

8. **Visualization:**

Creates and plots the evaluation matrix using Matplotlib.

In summary, the code employs XML comparison, WordNet-based semantic similarity, ontology reasoning, and graph isomorphism techniques to evaluate and compare two OWL files. The results are presented in a detailed evaluation matrix, and a bar chart provides a visual representation of the scores.

3. OWL – CSV Code:

Imports: The script begins by importing necessary libraries, including logging, pandas, re, base64, csv, and rdflib. These libraries provide essential tools for handling RDF data, logging information, and working with CSV files.

Global Variables: Two global variables are declared:

count: It keeps track of namespace counts.

uri: A global dictionary initialized to store prefixes and their corresponding namespaces.

Functions:

1. owl_csv_subclass(g, uri):

Purpose: Extracts subclass information from the RDF graph g and writes it to 'subclass.csv'.

Implementation: The function processes subclass relationships by querying the RDF graph for triples with the predicate 'rdf:type' and 'rdfs:subClassOf'. It then creates rows in the 'subclass.csv' file, capturing each subclass and its parent.

2. owl_csv_class(g, uri):

Purpose: Extracts information about OWL classes from the RDF graph g and writes it to 'owlclass.csv'.

Implementation: The function identifies OWL classes by querying the RDF graph for triples with the predicate 'rdf:type' and 'owl:Class'. It creates rows for each class in the 'owlclass.csv' file.

3. owl_csv_domain(g, uri):

Purpose: Extracts domain information from the RDF graph g and writes it to 'domain.csv'.

Implementation: The function processes domain relationships between properties and classes by querying the RDF graph for triples with the predicate 'rdfs:domain'. It creates rows for each relationship in the 'domain.csv' file.

4. owl_csv_range(g, uri):

Purpose: Extracts range information from the RDF graph g and writes it to 'range.csv'.

Implementation: The function processes range relationships between properties and classes by querying the RDF graph for triples with the predicate 'rdfs:range'. It creates rows for each relationship in the 'range.csv' file.

5. owl_csv_instances(g, uri):

Purpose: Extracts information about instances from the RDF graph g and writes it to 'instances.csv'.

Implementation: The function identifies instances of OWL classes by querying the RDF graph for triples with the predicate 'rdf:type' and the class type. It creates rows for each instance in the 'instances.csv' file.

6. owl_csv_subproperty(g, uri):

Purpose: Extracts subproperty information from the RDF graph g and writes it to 'subproperty.csv'.

Implementation: The function processes subproperty relationships by querying the RDF graph for triples with the predicate 'rdf:type' and 'rdf:subPropertyOf'. It creates rows for each relationship in the 'subproperty.csv' file.

7. owl_csv_inverseof(g, uri):

Purpose: Extracts inverse property information from the RDF graph g and writes it to 'inverseOf.csv'.

Implementation: The function processes inverse property relationships by querying the RDF graph for triples with the predicate 'owl:inverseOf'. It creates rows for each relationship in the 'inverseOf.csv' file.

8. owl_csv_allvaluesfrom(g, onproperty_dict, uri):

Purpose: Extracts information about the allValuesFrom property restriction and writes it to 'allvaluesfrom.csv'.

Implementation: The function processes 'allValuesFrom' relationships by querying the RDF graph for triples with the predicate 'owl:allValuesFrom'. It combines information from temporary CSV files 'allvaluesfrom1.csv' and 'allvaluesfrom_temp.csv'.

9. owl_csv_somevaluesfrom(g, onproperty_dict, uri):

Purpose: Extracts information about the someValuesFrom property restriction and writes it to 'somevaluesfrom.csv'.

Implementation: The function processes 'someValuesFrom' relationships by querying the RDF graph for triples with the predicate 'owl:someValuesFrom'. It combines information from temporary CSV files 'somevaluesfrom1.csv' and 'somevaluesfrom_temp.csv'.

10. owl_csv_maxcardinality(g, onproperty_dict, uri):

Purpose: Extracts information about the maxCardinality property restriction and writes it to 'maxcardinality.csv'.

Implementation: The function processes 'maxCardinality' relationships by querying the RDF graph for triples with the predicate 'owl:maxCardinality'. It combines information from temporary CSV files 'maxcardinality1.csv' and 'maxcardinality_temp.csv'.

11. owl_csv_firstrest(g, first_dict, uri):

Purpose: Extracts information about the first and rest properties and writes it to 'firstrest.csv'.

Implementation: The function processes 'first/rest' relationships by querying the RDF graph for triples with the predicates 'rdf:first' and 'rdf:rest'. It combines information from temporary CSV files 'first.csv' and 'firstrest_temp.csv'.

12. not_processing(g, uri):

Purpose: Logs any RDF triples not processed by other functions.

Implementation: The function captures triples that are not related to class, property, domain, range, instance, subclass, subproperty, inverseOf, allValuesFrom, someValuesFrom, maxCardinality, or first/rest properties.

Conclusion: The script successfully extracts and organizes essential information from an RDF ontology into CSV files, facilitating further analysis. The use of the rdflib library enables efficient querying of RDF graphs, and the modular structure of functions enhances code readability and maintainability. The generated CSV files serve as structured data sources for diverse applications, promoting better understanding and utilization of RDF ontologies.

Algorithm being used:

1. OWL Class Extraction (owl_csv_class):

Algorithm: Query the RDF graph for triples with the predicate 'rdf:type' and 'owl:Class'. Iterate through the results and extract information about each OWL class. Write the extracted information to the 'owlclass.csv' file.

How it works: The script uses the rdflib library to query the RDF graph for instances of the class 'owl:Class'. It then extracts information such as class names, labels, and comments and writes them to a CSV file for each class found.

2. Subclass Extraction (owl_csv_subclass):

Algorithm: Query the RDF graph for triples with the predicate 'rdf:type' and 'rdfs:subClassOf'.

Iterate through the results and extract information about each subclass and its parent. Write the extracted information to the 'subclass.csv' file.

How it works: The script identifies subclasses by querying the RDF graph for instances of 'rdfs:subClassOf'. It captures the subclass and its parent class, organizing this information into rows in the 'subclass.csv' file.

3. Property Domain Extraction (owl_csv_domain):

Algorithm: Query the RDF graph for triples with the predicate 'rdfs:domain'. Iterate through the results and extract information about each property and its domain (class). Write the extracted information to the 'domain.csv' file.

How it works: The script looks for triples with the 'rdfs:domain' predicate to identify relationships between properties and their domains (classes). It captures this information and stores it in the 'domain.csv' file.

4. Property Range Extraction (owl_csv_range):

Algorithm: Query the RDF graph for triples with the predicate 'rdfs:range'. Iterate through the results and extract information about each property and its range (class). Write the extracted information to the 'range.csv' file.

How it works: The script searches for triples with the 'rdfs:range' predicate to identify relationships between properties and their ranges (classes). It organizes this information into rows in the 'range.csv' file.

5. Instance Extraction (owl_csv_instances):

Algorithm: Query the RDF graph for triples with the predicate 'rdf:type' for each class.

Iterate through the results and extract information about each instance. Write the extracted information to the 'instances.csv' file.

How it works: The script identifies instances by querying the RDF graph for triples with the 'rdf:type' predicate for each class. It captures information about each instance and stores it in the 'instances.csv' file.

6. Subproperty Extraction (owl_csv_subproperty):

Algorithm: Query the RDF graph for triples with the predicate 'rdf:type' and 'rdf:subPropertyOf'. Iterate through the results and extract information about each subproperty and its parent property. Write the extracted information to the 'subproperty.csv' file.

How it works: The script identifies subproperties by querying the RDF graph for instances of 'rdf:subPropertyOf'. It captures information about each subproperty and its parent property, organizing it into rows in the 'subproperty.csv' file.

7. Inverse Property Extraction (owl_csv_inverseof):

Algorithm: Query the RDF graph for triples with the predicate 'owl:inverseOf'. Iterate through the results and extract information about each property and its inverse. Write the extracted information to the 'inverseOf.csv' file.

How it works: The script identifies inverse property relationships by querying the RDF graph for instances of 'owl:inverseOf'. It captures information about each property and its inverse, storing it in the 'inverseOf.csv' file.

8. Property Restriction: allValuesFrom, someValuesFrom, maxCardinality (owl_csv_allvaluesfrom, owl_csv_somevaluesfrom, owl_csv_maxcardinality):

Algorithm: Query the RDF graph for triples with the respective property restriction predicates.

Extract information about the property, its restriction, and related class. Combine information from temporary CSV files for each restriction.

How it works: The script searches for triples with 'owl:allValuesFrom', 'owl:someValuesFrom', and 'owl:maxCardinality' predicates. It processes the information and combines it from

temporary CSV files ('allvaluesfrom1.csv', 'somevaluesfrom1.csv', 'maxcardinality1.csv') into the final output CSV files.

9. first and rest Property Extraction (owl_csv_firstrest):

Algorithm: Query the RDF graph for triples with the predicates 'rdf:first' and 'rdf:rest'. Extract information about the first element, rest elements, and related class. Combine information from temporary CSV files.

How it works: The script searches for triples with the 'rdf:first' and 'rdf:rest' predicates. It processes the information and combines it from temporary CSV files ('first.csv', 'firstrest_temp.csv') into the final output CSV file.

10. Not Processing (not_processing):

Algorithm: Log any RDF triples not processed by other functions.

How it works: The script captures RDF triples that do not match the patterns processed by other functions. It logs this information for further investigation.

4. CSV – OWL Code:

Here's a breakdown of the key parts of the code:

1. Function Definitions:

a. csv_owl_final Function: This function takes various parameters representing different aspects of the ontology (e.g., subclasses, domain axioms, range axioms) and a graph (g) to build the ontology. The main steps include:

Class Definition (OWL Class): Iterates through the class information in the 'owlclass' sheet and adds class definitions to the graph.

Subclass Axioms (RDFS.subClassOf): Processes subclass information from the 'subclass' sheet and adds subclass relationships to the graph.

Domain Axioms (RDFS.domain): Processes domain axioms from the 'domain' sheet and associates classes with object properties.

Range Axioms (RDFS.range): Processes range axioms from the 'range' sheet and associates object properties with classes.

Individuals (OWL.NamedIndividual): Processes instances from the 'instances' sheet and adds individual instances to the graph.

Subproperty Axioms (RDFS.subPropertyOf): Processes subproperty information from the 'subproperty' sheet and adds subproperty relationships to the graph.

Inverse Axioms (OWL.inverseOf): Processes inverse property information from the 'inverseOf' sheet and adds inverse property relationships to the graph.

SomeValuesFrom Axioms (OWL.someValuesFrom): Processes 'someValuesFrom' information and adds corresponding axioms to the graph.

AllValuesFrom Axioms (OWL.allValuesFrom): Processes 'allValuesFrom' information and adds corresponding axioms to the graph.

MaxCardinality Axioms (OWL.maxCardinality): Processes 'maxCardinality' information and adds corresponding axioms to the graph.

First/Rest Axioms (RDF.first/RDF.rest): Processes 'firstrest' information and adds corresponding axioms to the graph.

Serialization: Serializes the graph into an OWL file named 'output.owl' in XML format.

b. main Function: The main function reads data from an Excel file and calls the csv_owl_final function with the relevant parameters.

Additional Notes:

The code assumes the presence of specific sheets ('owlclass', 'subclass', etc.) in the Excel file, and the structure of the data in these sheets determines the ontology structure. The default namespace for the ontology is set to 'http://www.semanticweb.org/rohitbhatia/ontologies/2023/3/ontology#' if no prefix/namespace information is provided. The code generates RDF triples representing classes, properties, individuals, and axioms and then serializes them into an OWL file.

5. Count:

This Python script processes a collection of OWL ontology files, extracts specific information from them, and then generates a CSV file containing the count of different axioms present in these ontologies. Here's a breakdown of how the code works:

1. Setup:

```
import os
from rdflib import Graph
import csv
from funowl.converters.functional_converter import to_python
```

The script imports necessary modules, including os for working with the filesystem, rdflib for handling RDF data, csv for CSV file operations, and funowl for working with OWL ontologies.

2. File Processing:

```
f1 = open('/home/rohit19268/test/didntwork.csv', 'w')
dir_path = '/home/rohit19268/test/files'
files = os.listdir(dir_path)
total = 0
axioms_dict = {}
```

It initializes a file (f1) to log any files that couldn't be processed and defines a directory (dir_path) containing OWL ontology files.

3. Iterating Over Ontology Files:

```
for file_name in files:
    file_path = os.path.join(dir_path, file_name)
    try:
        if os.path.isfile(file_path):
            internal_file = to_python(file_path)
            g = Graph()
            internal_file.to_rdf(g)
            s1 = file_path.rsplit('_')[-1]
            s2 = s1.split('.')
            s = '/home/rohit19268/test/files' + 'ore' + s2[0] + '.owl'
            g.serialize(destination=s, format='xml')
```

The script iterates over each file in the specified directory.

It attempts to convert the ontology file (file_path) to a Python internal representation using funowl and then serializes it to an OWL file (s) in XML format.

4. Extracting Axioms:

```
for stmt in g:
    s = stmt[1]
    if '#' in s:
        index = s.index('#')
        s1 = s[index+1:]
        if s1 != 'type':
            axioms_dict[s1] = axioms_dict.get(s1, 0) + 1
            total += 1
```

It iterates over RDF triples in the graph and extracts the predicate (property) from each triple.

If the predicate contains a '#' character, it extracts the local name after the '#' and counts it as an axiom (excluding those with the name 'type').

5. Logging Unprocessed Files:

```
except Exception as e:
    writer = csv.writer(f1, lineterminator='\n')
    writer.writerow(s2[0] + '.owl')
    writer.writerow("")
```

If an exception occurs during file processing, it logs the file name in the 'didntwork.csv' file.

6. Writing Axiom Counts to CSV:

```
fields = ['Axiom', 'Count']
s = '/home/rohit19268/test/count_ontology_new.csv'
f = open(s, 'w')
writer = csv.writer(f, lineterminator='\n')
row = ["Total axioms for ontologies", total]
writer.writerow(row)
writer.writerow("")
writer.writerow(fields)
for key, value in axioms_dict.items():
    percent = (value / total) * 100
    row = [key, value, percent]
    writer.writerow(row)
```

It writes the total axiom count, and individual axiom counts along with their percentages to a CSV file ('count_ontology_new.csv').

7. Closing Files:

```
f1.close()
f.close()
```

It closes the opened files.

8. Additional CSV Output:

```
fields = ['Axiom', 'Count']
s = '/home/rohit19268/test/count_ontology.csv'
f = open(s, 'w')
writer = csv.writer(f, lineterminator='\n')
row = ["Total axioms for ontologies", total]
writer.writerow(row)
writer.writerow("")
writer.writerow(fields)
for key, value in axioms_dict.items():
    percent = (value / total) * 100
    row = [key, value, percent]
    writer.writerow(row)
```

It repeats a similar process to generate a different CSV file ('count_ontology.csv').

Summary:

The code processes a collection of OWL ontology files, extracts axioms from them, and generates CSV files with axiom counts and percentages. The specific axioms considered are those extracted from the predicates in the RDF triples of the ontologies.