INM713 Semantic Web Technologies and Knowledge Graphs
Part 2

Konstantinos Meli & Quang Huy Bui
MSc Data Science
City University of London
konstantinos.meli@city.ac.uk & quang.bui.2@city.ac.uk

Task 1: Tabular Data to Knowledge Graph (Task RDF)

Subtask RDF.0

For this task, we added two restaurants, each with two menu items, for a total of four additions. We recorded all changes in the `cw_data` file.

Subtask RDF.1

Since entity URIs cannot contain spaces, we replace spaces with underscores in the entity names. We then concatenate these entity names with the namespace.
In addition, we require identifiers for certain fields:

- `address_id` is a combination of "address", "city", and "state".
- `restaurant_id` is constructed from the address_id and the "name" of the restaurant.
- `item_value_id` is a concatenation of "item value" and "currency".

Note that menu items sold in different restaurants have different IDs because they may not be the same because each restaurant has its own recipe. Therefore, `item_id` combines "menu item" and `restaurant_id`.

Subtask RDF.2

In this task, we developed a method to link rows of items to their corresponding classes. We combined certain fields to create entity IDs, as described above.
Additionally, we ensured that we followed the namespace and property naming conventions outlined in `cw_onto` for our queries.
It is important to note that we mapped entities with `namespace:entity_id` and created a name for them by parsing the string for the respective column.
This approach allowed us to call `?name a cw:name` in our queries to retrieve the relevant data instead of returning the URI for the entity.

Subtask RDF.3

We used the OWL 2 RL library implementation called `owlrl` to perform reasoning for this task. To do so, we loaded our `Graph()` entity with `cw_onto` and called `expand()` on the graph, which is the combination of the generated ontology with `cw_onto`.

Subtask RDF.4

We have implemented a mechanism for reusing information about external URIs in our graph. This mechanism allows us to query external knowledge graphs and generate triples.

To enable this functionality, set `enable_external_uri: bool = False` to `True`. We have also set a threshold for external URI matches. If the vocabulary similarity score is lower than the threshold, which indicates that the query may not return a good match for the requested vocabulary, we create the entities using our internal `namespace`. This ensures the entities' meaning.

You can adjust the threshold with `external_uri_score_threshold = float`. Note that we sometimes disable external queries manually to reduce unnecessary API calls. We do this, for example, when dealing with entities that have IDs.

Subtask RDF.5 (Optional)

We used external knowledge graphs (KGs) to perform logical tasks but did not use them to correct location names. The `description` column in this data contains a mix of normal pizza descriptions and some ingredients. To distinguish between these, we utilised external KGs. If external KGs are disabled, the ingredient data in the description column will not be correctly classified.

Enabling the use of external knowledge graphs can help classify data in cases where the description column contains a mixture of normal pizza descriptions and ingredients. For example, a `description` like "Olives, onions, capers, tomatoes" can be classified and represented as four new triples such as `cw:item cw:hasIngredient dbr:onion` using our mechanism. The mechanism involves splitting the input string into individual ingredients, removing trailing "s" at the end of each element, and searching for these ingredients on public knowledge graphs. Therefore, it can perform queries concisely.

In another case, when the description is "Choose a pizza size", it will be stored as a `cw:description` triple.

Task 2: SPARQL and Reasoning (Task SPARQL)

In this task in order to move on to the subsections of this part, a class `SparqlQuery`. This class checks that the ontology passed is in the correct format `.ttl` or `.rdf`. It also initialises the rdflib.graph of the ontology that gets passed through the initialisation of the class. The class contains a single function. The function `make_query_to_csv` does exactly as it says by querying the SPARQL queries through the graph in order to pass the results from it to the .csv file that is needed.

Subtask SPARQL.1

For the first SPARQL task a simple filtering query is to be made. Therefore a query is done where all the restaurants that exist in the state of Texas are returned. After using the appropriate triples for our

columns to be set up correctly in the csv. For example to set-up the state variable as `stateName` we first extract the address line from the `cw:locatedAddress` into `address`. After the address is set up we can extract different information about the address like cityName, and stateName by doing this for example `?address cw:locatedState ?state` and then to get the name from the URI you `?state cw:name ?stateName`. The `FILTER (?stateName = "TX")` is used after all the variables for the query are set in order to filter all the other states except Texas out of the csv. The csv returns the name, address, city and state of each restaurant.

Subtask SPARQL.2

For the second subtask the average and filter function was used. In order to use both those functions the query will return the average price of all items on the menu of a specific restaurant. In order to do that, we set up a variable to save the avg values in by doing `(AVG(?menu_item_price) AS ?avg_value)` and filter by using `FILTER(?restaurantName = "Burgers & Cupcakes")`. The same is done as in subtask one to extract the names of each of the restaurant names and then filtering only through the Burgers & Cupcakes restaurant.

Subtask SPARQL.3

For this task the query aggregates and filters the results and then are grouped. To achieve this the number of restaurants in a city (aggregate) is returned and is filtered only to show the cities that are not in Washington and is grouped by the cityName. In order to aggregate the restaurant number when setting up the variables the `COUNT` function is used like so `(COUNT(?restaurants) AS ?num_restaurants)`.

Subtask SPARQL.4

In this task a query is created where the results are aggregated, filtered and then are grouped. The results have to be ordered according to two variables. So what is done in this query it returns the cities with a number of restaurants higher than 7 AND have an average value on their menus of higher than 10 USD. They are ordered in a descending order of the number of restaurants in each city along with the average value. In order to filter this much information, this time instead of using the `FILTER` function the `HAVING` function is used as so `HAVING (COUNT(?restaurant) > 7 && AVG(?menu_item_price) > 10)` and to order the results the `ORDER` function is used like so `ORDER BY DESC(?num_restaurants) ?avg_value`.

Subtask SPARQL.5

For the final subtask of this part uses the `UNION` graph pattern and negation, so the query created returns the names of the restaurants that are either in New York City or don't have any items on the menu worth higher than 5.

Task 3: Ontology Alignment (Task OA)

For this task basic alignment was performed between the `pizza.owl` ontology and the `cw_onto` ontology. This means that equivalent classes and properties are identified between the two ontologies compared. This is done so that SPARQL queries can be performed with the vocabulary of one ontology on to the other.

Subtask OA.1

In order to save equivalences between the two ontologies mentioned above each ontology is parsed into its own unique graph from the `rdflib` library. Using those graphs, for loops are created for each graph passing through each entity of the graph and string values are retrieved by using the said entity and using the label predicate which uses the URI reference for the rdf label `http://www.w3.org/2000/01/rdf-schema#label`. The two strings are then compared in order to get a similarity score using the jaro-winkler function from the Levenshtein library. Then the entities of the strings that had a similarity score of more than 0.9 are added onto an alignment graph created that is then extracted as the "equivalences.ttl" file giving us the results.

Subtask OA.2

In this simple task in order to perform reasoning from all the sources mentioned, then all the sources need to be parsed into their own unique graphs. Then those graphs are added onto a newly created graph in order to load all of them in a single file. To perform reasoning on all these sources the OWL RDFS and our created ontologie's namespace is bound on to this graph.

Task 4: Ontology Embeddings (Task Vector)

Subtask Vector.1

We merged the `cw_onto` and our generated graphs to meet the requirements and applied vectorisation to the new graph.

We also ran `OWL2Vec*` on three different configurations, which can be found in the `./task_vector/owl2vec_config/` directory. These configurations are modifications of the `default.cfg` file, focusing on changing the reasoner type, walker configuration, and training iterations.

Lastly, we saved the resulting binary and `.txt` files to `./owl2vec_embeddings/config_{}_output/.`

Subtask Vector.2

We have created a `helper.py` file to facilitate this task. The file contains the following functions:

- `generate_word_model`: This function finds the most similar vocabulary from the `OWL2Vec*` model generated in the previous task and returns its vectorisation score. It also returns the words so they can be labelled.
- `dim_reduction`: This function performs Principal Component Analysis (PCA) on the vectorised model. Currently, the default number of dimensions is set to 2, allowing it

to be visualised in 2D graphics. However, the function can be modified by passing the desired number of dimensions to the `_n_dim` parameter.

- `visualise_2d`: This function generates 2D graphs of the dimension-reduced data generated in the previous step using `dim_reduction`. If we change the target dimension from the previous step to more than 2, we must modify this function or create a new one. For example, if we modify the number of dimensions to 3, we must create a function to generate a 3D graph.

We compared multiple pairs of vocabulary to generate an overall view. Three pairs were selected for each comparison:

- High similarity: We observed that "pizza" has a high similarity score with "pepperoni", "mushroom", and "cheese", as all of them are ingredients in pizza. However, "pasta" has a high dissimilarity score in comparison.
- High dissimilarity: The graph showed that "rainbow", "sushi", "cocktails", and "spot" have significant dissimilarity scores. In contrast, "Blockbuster" and "Newport" have a close similarity score, indicating that they might be locations with similar names.