Semantic Web Technologies & Knowledge Graphs Coursework Report

Ollie Keers

Introduction

This report accompanies work covering the transformation of a dataset into a knowledge graph, and associated analysis. The dataset in question is a Kaggle dataset about pizza restaurants and the items that they sell, https://www.kaggle.com/datafiniti/pizza-restaurants-and-the-pizza-they-sell.
This work is structured into 5 sections:
OWL: the construction of an ontology to represent the domain of the dataset.
RDF: the transformation of the csv file into triples to construct a knowledge graph.
SPARQL: using reasoning and executing indicative queries on the constructed knowledge graph, to extract information representative of that an end-user may wish to retrieve.
OA: the alignment of the constructed ontology with the well-known pizza.owl ontology.
Vector: the creation of embeddings for the knowledge graph, with subsequent analysis to investigate how data properties have been captured.

OWL

Examination of the dataset provided indicates that it contains four broad classes of information, which have been treated as the superclasses in this ontology, namely:

1. Place - information relating to the location of establishments.
2. Establishment – information relating to the nature of the business and services offered.
3. Food – information relating to consumable items.
4. Currency – information about the monetary system.

These four classes are entirely distinct and should contain no overlapping entities, they are therefore treated as disjoint. Besides currency, which is a self-contained concept within this ontology, each of these classes has several sub-classes which we discuss in turn.

Within the superclass of ‘Place’, we have location information at hierarchical specificity. A ‘Country’ contains several ‘States’, which contain one or more ‘Cities’, containing multiple ‘Postcodes’ which in turn contain multiple ‘Addresses’. While this information is hierarchical, these are not hierarchical classes – Cities are not a subclass of States, for example, because a City is not a type of State. While a particular location is within both a city and a state simultaneously, these are separate concepts. The 5 location sub-classes are therefore disjoint from each other, but the transitive object property ‘locatedIn’ preserves their hierarchical relationship. This transitivity is intuitive – if a postcode is locatedIn a city, and a city is locatedIn a state, that postcode is locatedIn a state. Other asymmetric and irreflexive object properties also capture aspects of this through range and domain restrictions. One such example is inCity which can have domains of Establishment, Address, or Postcode (but not State or Country as neither of these are contained within a city), and must have a range of City. These properties are irreflexive (a City cannot be in itself), and asymmetric (if a City is locatedIn a State, the State cannot be locatedIn the City). If these properties only applied to other Places, they would be functional (a Postcode exists in only one State). However, as they also apply to Establishments (which may have more than one physical location), this is not the case.

Establishments are a class of entities disjoint with Places. These represent the fictional, abstract concept of a business, which may have more than one set of premises associated with it. It is therefore possible that an Establishment may have more than one address, and be located in more than one state (such as Papa John’s which has several stores). There are many different types of Establishment, extending far beyond the scope of the dataset provided (in accordance with the open world assumption). Two of these establishment categories have been explicitly created to denote the services that are offered therein: DineIn and TakeAway, where products are available for consumption either on or off the premises respectively. It is entirely possible for a single

establishment to offer both DineIn and TakeAway services. There are many other subclasses of establishment, such as a Pizza Restaurant, an Italian Restaurant, a University, or a Deli. These further subclasses of establishment may have many members or just one. Again, establishments may be members of more than one of these subclasses. For the purposes of this dataset, each Establishment sells Menu Items.

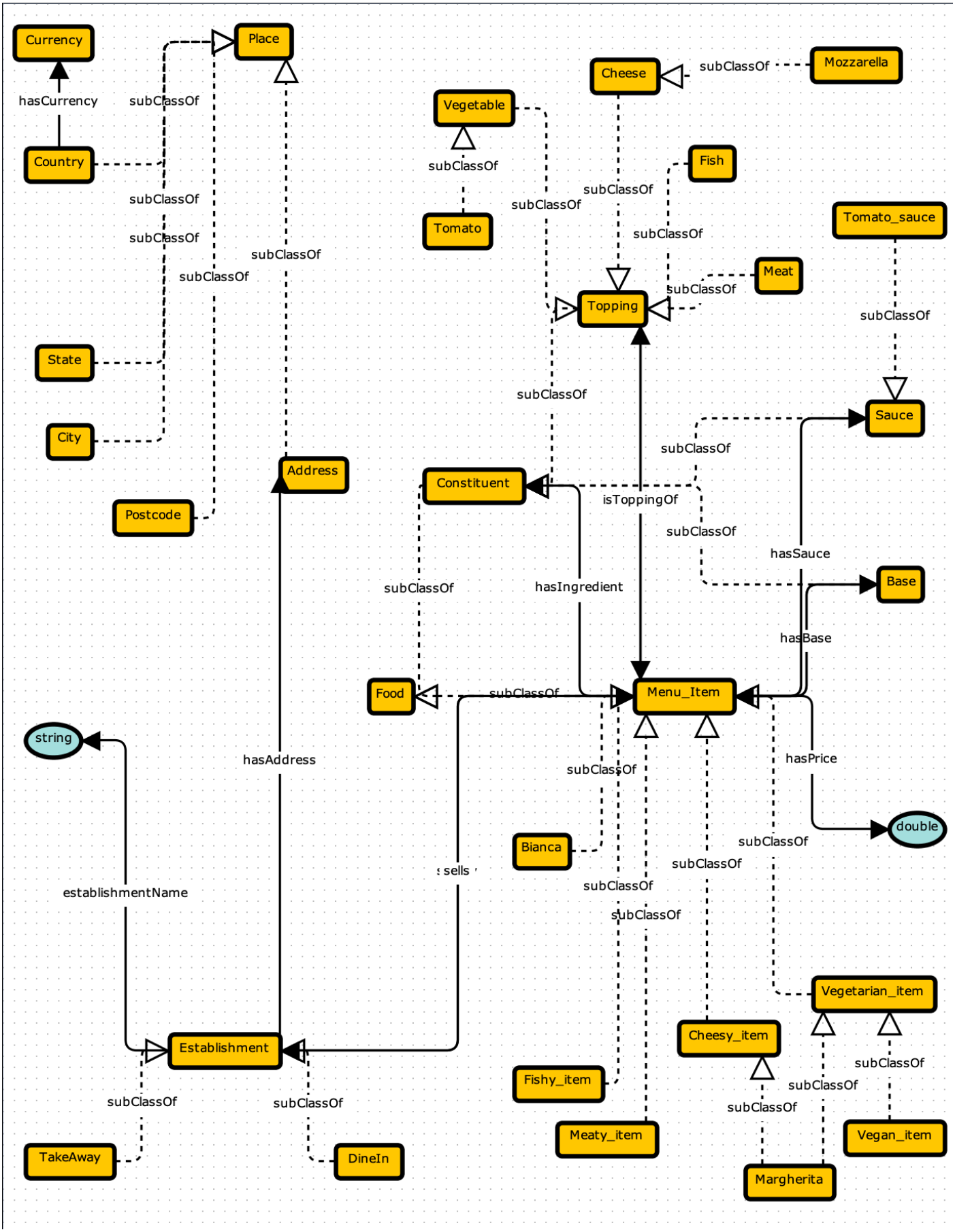
Menu Items are a subclass of Food, representing items within the dataset that are available for purchase. As each menu item is being sold, it must have a price associated with it, and so this is a required condition of being a menu item rather than just 'Food'. The other major subclass of Foods defined here is Constituents, where each is an ingredient or part of at least one menu item. This is a transitive property, as an ingredient of an ingredient of a menu item, is an ingredient of that menu item – if peppers are an ingredient in the sauce of a pizza, they are an ingredient of the pizza. It is possible for a constituent to be a menu item if it is also available for purchase (it may be possible to purchase pizza sauce from an establishment that is a deli, for example) and so Menu Item and Constituent are not disjoint. A menu item may have multiple constituents, or none.

Constituents have been broken into 3 subclasses: bases, toppings and sauces. These are disjoint. There are several possible bases (thick, thin, square etc.) and sauces (tomato, bbq, curry). Toppings can be further subdivided into the disjoint classes cheese, meat, fish and vegetable. Vegetable has been defined to be any plant-matter, and so incorporates fruits such as tomatoes. Meat and fish have been treated as the parts of land- or water-based creatures respectively, to allow for the dietary preferences of pescatarians. These classes are therefore disjoint within this ontology, whereas fish may be considered as a subclass of meat in other ontologies where meat represents *any* animal part.

Menu Items have subclasses based upon their constituents. A cheesy item, meaty item, or fishy item contain cheese, meat, or fish respectively. It is possible for a menu item to occupy more than one of these classes, such as a pizza with meat and cheese. A vegetarian item is an item that contains neither meat nor fish and is therefore disjoint with meaty & fishy items. A vegan item is a subclass of a vegetarian item that also contains no cheese, and is thus disjoint with cheesy items. A Bianca has been defined as a menu item that contains no tomato sauce or tomato, but with no other restrictions. A margherita has been defined as a menu item that only has tomato sauce, and only has mozzarella as a topping, but no restrictions have been placed upon its base. For the purposes of this ontology, a pizza that also contains whole tomatoes, or basil, would not be considered a margherita.

The final class, currency, represents the legal tender of a particular country. This abstract concept is disjoint from all other classes.

A schematic representation of the ontology, generated using CoModIDE (https://comodide.com), is given overleaf. It is worth noting that the majority of the ontology is given over to modelling Food (right hand side), with Place (upper left), Establishment (bottom left), and Currency (uppermost left) representing far less of the modelled domain.


Currency
Place
hasCurrency
subClassOf
Country
subClassOf
subClassOf
subClassOf
subClassOf
State
City
Postcode
Address
Cheese
subClassOf
Mozzarella
Vegetable
subClassOf
Tomato
subClassOf
subClassOf
Fish
subClassOf
Meat
subClassOf
Topping
Tomato_sauce
subClassOf
Sauce
subClassOf
subClassOf
Constituent
isToppingOf
subClassOf
hasSauce
Base
hasBase
subClassOf
hasIngredient
Food
subClassOf
Menu_Item
string
hasAddress
subClassOf
hasPrice
double
Bianca
subClassOf
subClassOf
subClassOf
subClassOf
establishmentName
Establishment
Vegetarian_item
Cheesy_item
subClassOf
subClassOf
subClassOf
Fishy_item
Meaty_item
Vegan_item
Margherita
subClassOf
subClassOf
TakeAway
DineIn

RDF

For the purposes of this task, each row represents a unique entity, containing a menu item that is for sale by an establishment for a particular price. That particular entity may be a unique instance of a common subclass (the pepperoni pizza sold by restaurant X for $Y), or something that is entirely unique to that establishment. Initial extraction techniques for naming the URI for each of these entities led to problems further through the work, due to non-unique naming. 'ok:establishmentName_productName' caused clashes for identical items sold by multiple branches of the same chain (e.g. ok:PapaJohns_PepperoniPizza'), which did not have consistent prices or information across branches. 'ok:{rowIndex}' caused clashes with postcodes, which are 5 digit numbers, which led to significant issues due to class disjointedness between Food and Place. This led to the decision to use the row index, with a prefix to generate the URI for each of these entities, in the form "http://www.city.ac.uk/ds/inm713/ollie_keers/product_{rowIndex}".

This initial node is a named individual, said to be for sale at the establishment in the 'name' cell for that row. The product is defined to be a member of a class given by the 'menu item' name (e.g. product_123 a pepperoni_pizza), which is a sub-class of Menu Item. For the purposes of the subsequent SPARQL tasks, where an item's name indicates that it is a margherita or Bianca, triples indicating this are explicitly added. Some items have further information provided in 'item description', but this is specific to the product rather than the broader class of that item (such as number of toppings on a 'create your own' pizza which varies considerably) and so this is stored as a string for that particular product node. There is very little consistency to the nature of the information that has been provided in this cell, so extracting ingredients, for example, is not possible in the scope of this work. Much like the descriptions, the 'price' of an item does not apply to the class as a whole, and so is stored as a literal for that product node only.

The establishment has been given a URI generated from its name, and its name is also stored as a string. Location information has been taken from several columns of the table, and associated triples created as per the ontology. As mentioned previously, an establishment represents the abstract concept of a business and so may have more than one physical location. Equally, it is possible for multiple businesses to be based out of the same premises (such as a ghost kitchen, or a combination restaurant and deli). The associated address has thus been used to generate a URI, and if duplicated by later establishments this URI will be associated with all of them. Further information about the establishment is given in 'categories' as a list of strings, which are treated as 'hashtags' in construction of these URIs, such as #pizza_place, or #italian_restaurant, each of which is a subclass of establishments. This has been used not only to avoid potential clashes with establishment names, but also to reflect the common usage of hashtags to provide descriptors online. As defined in our ontology, where these hashtags include the term 'restaurant' we assume that they offer dine-in, and where 'take' is included, we assume they offer take away and encode these as further class memberships.

There are clearly errors in the data that has been entered in the city and state columns, from inspection of the csv file. This may cause issues where the entries in the 'city' and 'state' column are the same (or highly similar when querying dbpedia using lexical matching). This may cause entities to have both type City and type State. Postcodes are theoretically unique to that area, but if they were not recorded correctly this could also lead to conflicts.

Querying the DBPedia endpoint to enable the reuse URIs is a lengthy process, with information for city, country and state sought for each row. All of the restaurants are in the US, and so duplicating this query is very inefficient. There are only 50 states in theory, and it is likely that multiple restaurants (or at least rows) will exist for each city. In order to expedite the querying process, empty dictionaries were created, and each URI was added when retrieved. This dramatically cut down on computation time and was an appropriate technique, given the dataset size. For a larger knowledge graph, hashing may be useful to further expedite the querying process.

SPARQL

Programmatic reasoning was performed to expand the graph, using OWL RL semantic deduction.

SPARQL.2

```
18      qres = g.query(
19      """SELECT ?name ?address ?city_name WHERE{
20     ?entity a ok:Bianca ;
21            ok:soldBy ?establishment .
22     ?establishment ok:establishmentName ?name ;
23                   ok:hasAddress ?add ;
24                   ok:inCity ?city .
25      ?add ok:addressName ?address .
26      ?city ok:cityName ?city_name .
27      }""")
```

Line 19: Query returns name, address, and the name of the city.
Line 20: Selects all entities that are of type ‘Bianca’
Line 21: Selects establishments that sell those entities
Line 22: Selects the string containing the establishment name
Line 23: Selects the address of that establishment
Line 24: Selects the city of that establishment
Line 25: Selects the string containing the written address of the establishment
Line 26: Selects the string containing the name of the city.

```
30      f_out = open('query1.csv',"w+")
31      for row in qres:
32             #Row is a list of matched RDF terms: URIs, literals or blank nodes
33         print(row.name, row.address, row.city_name)
34         line_str = '\"%s\",\"%s\",\"%s\"\n' % (row.name, row.address, row.city_name)
35         f_out.write(line_str)
36
37      f_out.close()
```

Line 30: opens the output csv file
Line 31: loops through the query results
Line 33: prints the results, for convenience
Line 34: constructs a comma-separated string of results, order: restaurant name, address, city name
Line 35: writes this line to the output csv file
Line 37: closes the output file

SPARQL.3

```
49      qres = g.query(
50      """SELECT ( ROUND(100*AVG (?x))/100 AS ?average) ?cc
51      WHERE{
52     ?entity a ok:Margherita ;
53          ok:hasPrice ?x .
54     ?entity ok:soldBy ?est .
55     ?est ok:inCountry ?country .
56     ?country ok:hasCurrency ?currency .
57     ?currency ok:currencyCode ?cc .
58
59      }""")
60
61
62
63      for row in qres:
64          #Row is a list of matched RDF terms: URIs, literals or blank nodes
65          print("Average price for a Margherita is %s %s" % (str(row.average), str(row.cc)))
```

Line 50: AVG gives the average of ?x, but in order to return a price average to a sensible number of decimal places, ROUND has been used. This rounds to the nearest integer, so this has been multiplied by 100 before rounding, and divided by 100 afterwards to give two decimal places, corresponding to a price in cents. The currency code has also been given to show that it is in dollars.
Line 52: selects all entities that belong to the class ‘Margherita’
Line 53: selects the prices for these entities
Line 54:57 selects the places that sell these entities, the countries they are in, the currency used in that country, and the string representing that currency.
Line 65: prints the average price (to the nearest cent) and currency code.

SPARQL.4

```
77      qres = g.query(
78      """SELECT ?city_name ?state_name (COUNT (?x) AS ?howmany)
79      WHERE{
80      ?x a ok:Establishment ;
81           ok:inCity ?city ;
82           ok:inState ?state .
83      ?state ok:stateName ?state_name .
84      ?city ok:cityName ?city_name .
85       }
86       GROUP BY ?city
87       ORDER BY ?state_name ?howmany
88       """)
89
90
91
92       for row in qres:
93           #Row is a list of matched RDF terms: URIs, literals or blank nodes
94           print("%s, %s has %s restaurants" % (str(row.city_name), str(row.state_name), str(row.howmany)))
```

Line 78: Selects the city name, state name, and a count of how many establishments.
Line 80:82: Selects all establishments, and their respective cities and states
Line 83: Selects the state name
Line 84: Selects the city name
Line 86: Aggregates the count of establishments by city
Line 87: orders the results by the name of the state (in alphabetical order), and subsequently by number of establishments (ascending order)
Line 94: Prints details of the results of the query

SPARQL.5

```
107      qres = g.query(
108      """SELECT ?x
109      WHERE {
110          ?x a ok:Establishment .
111          FILTER NOT EXISTS { ?x  ok:inPostcode ?any .}
112      } """)
113
114      for row in qres:
115          #Row is a list of matched RDF terms: URIs, literals or blank nodes
116          print(row)
```

Line 108: Selects the establishment URI
Line 110: Selects all establishments
Line 111: Filters these establishments to those who do not have any objects to the inPostcode predicate – i.e. those with missing postcode.
Line 116: prints this information

OA

OA.1

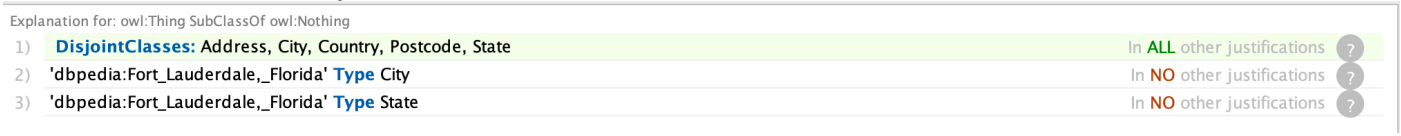
Equivalences were computed using lexical similarity, where the Levenshtein distance for the two terms is divided by the length of the longer term. Identical terms will therefore have a value of 0, and entirely dissimilar terms will have a value of 1. For this alignment, terms with a dissimilarity of below 0.3 (30% dissimilarity), are considered to be equivalent. This threshold is quite lenient, with a view to capturing as many terms as possible, but this has led to one significant error: isIngredientOf and hasIngredient are considered equivalent. These terms are the inverse of each other, but due to the length of the string, have a small dissimilarity value.

OA.2

The only two unsatisfiable classes when using HermiT are the two that were pre-existing in pizza.owl: CheesyVegetableTopping and IceCream.

OA.2.b

The generated data was loaded into Protégé with the aligned ontologies, and HermiT reasoner run for 5 hours, and was then stopped before it had completed explaining. In that time, 21 unsatisfiabilities appeared, all due to disjoint classes City and State, and the inconsistent data entry in the csv file. There are cases where for different establishments, the same location is given as both classes of location, such as for Brentwood. This results in triples Brentwood a City and Brentwood a State, which is not satisfiable with the assertion that these classes are disjoint. It is possible that endpoint querying with approximate matching has exacerbated this problem, as slight differences in name may have returned the same URI.



Programmatic OWL2RL reasoning was performed across the 2 ontologies, alignment, and generated data. Queries were run to see if any entities were a) a pizza:MeatyPizza, b) a ok:Meaty_Item, and c) a ok:Margherita. While the last of these was successful, there were no entities that were of the other two types.

Inspection of the equivalences.ttl file indicates that there is no triple stating that ok:Meaty_Item owl:equivalentClass pizza:MeatyPizza. This is due to the string similarity method used (Levenshtein distance as a proportion of word length). Because my ontology uses the term ‘item’ rather than ‘pizza’ this means that around half of the term has changed, well above the threshold of 0.3. Such a triple does exist for that ok:Vegetarian_Item owl:equivalentClass pizza:VegetarianPizza, because of the greater term length. The lack of equivalence triple would, on its own, result in no matching entities.

The lack of results for ok:Meaty_Item, conversely, indicates that even if the equivalence triple were present, no items would be returned. This is because during the extraction of triples, no information was provided regarding the nature of foods (whether meat, vegetable, dairy etc.). The created ontology did not focus on the taxonomy of foods and so while there may be a ‘chicken pizza’ or pizza who has a description ‘extra pepperoni’ there is nothing to indicate that chicken or pepperoni are meats, and thus that they represent ok:Meaty_Items.

These two factors, namely the lack of equivalence triple and lack of information regarding the nature of foods, have led to nothing being a pizza:MeatyPizza.

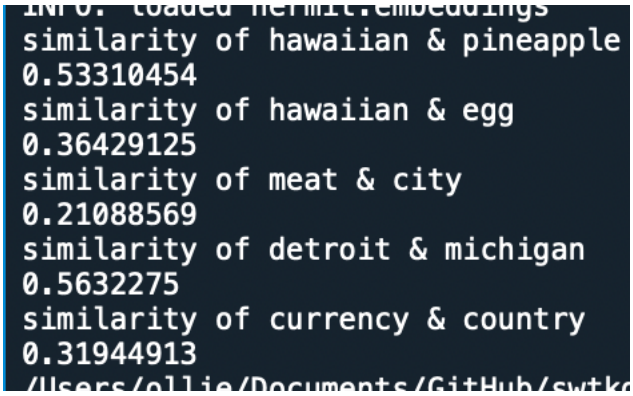
Vector

Vector.1

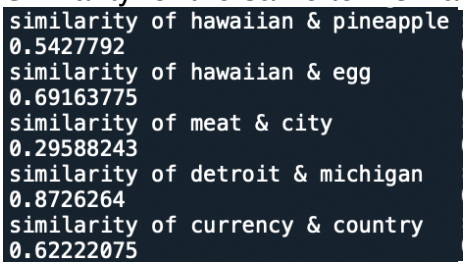
Three configurations of OWL2Vec* were used: default (default settings), elk (with elk reasoner) and hermit (with hermit reasoner, larger walk depth, and more training cycles).

Vector.2

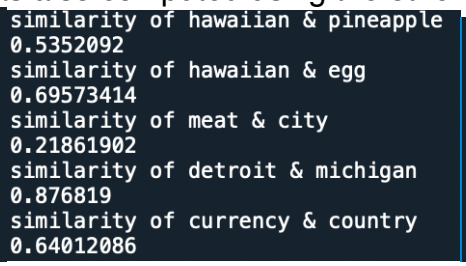
Five pairs of entities were selected: Hawaiian & pineapple (due to the presence of pineapple on a Hawaiian pizza); Hawaiian & egg (due to the absence of egg on a Hawaiian pizza); meat & city (these terms have no real relationship), Detroit & Michigan (Detroit is in Michigan), and currency & country (closely related but distinct terms). We do find that the entities that have a close relationship have the highest cosine similarity scores (Hawaiian & pineapple and Detroit & Michigan). Neither of these relationships were explicitly encoded from the ontology, but have been generated from the csv file transformation, so this similarity is encouraging. Lower similarities are seen for other concepts, which can be understood due to their being related but not directly interacting (such as Hawaiian and egg – both foods, but not found together). The lowest similarity is for meat & city, which is what we would expect to find given that the terms are not related.



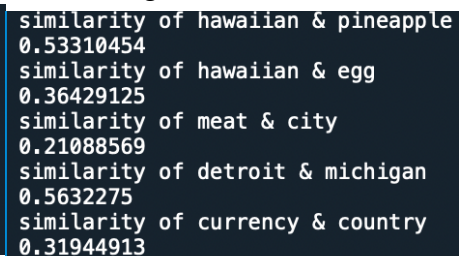
Similarity for the same terms was also computed using the other two configurations for OWL2Vec*



Default



Elk



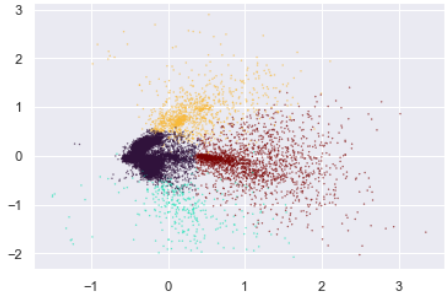
HermiT

It is interesting to note the differences in similarity scores.
Hawaiian & pineapple are have similar scores across all 3 configurations.
Hawaiian & egg are found to be more dissimilar for HermiT, which was trained for longer, than the other two configurations. These configurations of OWL2Vec* gave this pair of terms a higher similarity score than Hawaiian & pineapple.
Meat & city have low similarities across all 3 configurations, but are found to be more similar using the default OWL2Vec* configuration.
Detroit and Michigan are found to be very similar indeed for the default and Elk configurations, but much less similar with the most trained model.
Currency and country are also found to be much less similar with the most trained model.

For every pair of terms, the HermiT configuration has provided the lowest similarity scores. This would seem to indicate that the greater training, walk depth, and use of HermiT reasoner is accentuating the differences in terms, compared to the other two configurations. Both default and Hermit have found ‘egg’ to be more similar to Hawaiian than pineapple, which should not be the case. On balance, HermiT appears to provide the most balanced similarity scores.

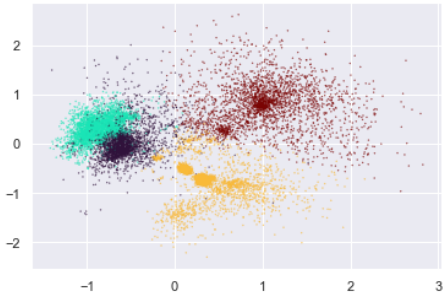
Vector.3

Clustering was also performed for all 3 configurations, with n clusters from 1:10 attempted. The optimal number of clusters, judged by eye, for configuration is each shown below.



Default, k=4

Elk, k=5

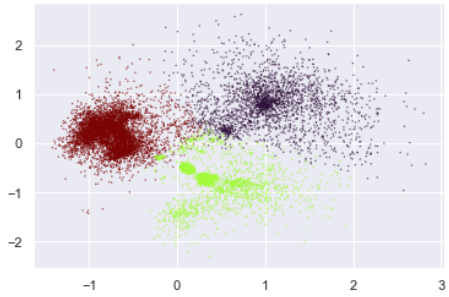


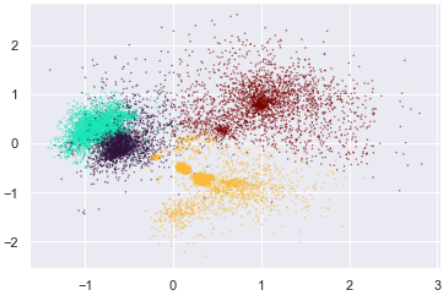
HermiT, k=4

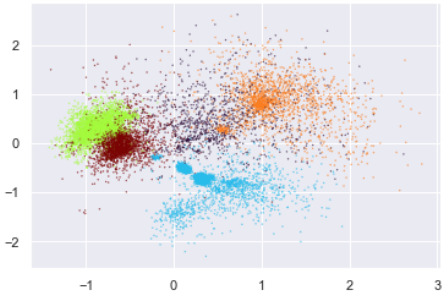
Cluster membership was found to be as outlined in the table below:

Cluster	Default	Elk	Hermit
0	*Multiple classes present*	Establishment descriptors	Places
1	Places	*Multiple classes present*	Establishment Descriptors
2	Establishment descriptors	Establishment descriptors	Products
3	Food	Places	Food
4	-	Food	-

Because the optimal class distinctions were found for the HermiT configuration in the last task, these clusters were chosen for treatment in greater depth. Images for k=3, 4 & 5 for this configuration are provided below, to illustrate why k=4 was chosen as the optimal number of clusters. K=5 begins to split areas of density spuriously, without selecting the areas that seem most distinct to a human, such as the dense plots at around (0.25, -0.5).

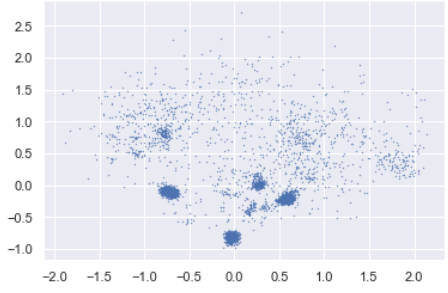






Each of the clusters for k=4 was extracted and plotted individually, and its membership examined.

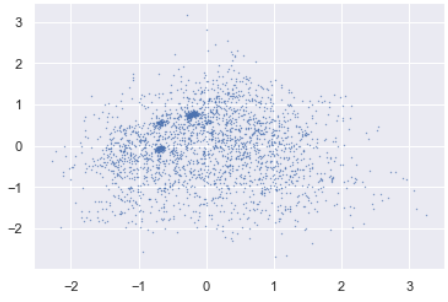
Cluster 0:



http://www.city.ac.uk/ds/inm713/ollie_keers/55331	-0.598011	...	0
http://www.city.ac.uk/ds/inm713/ollie_keers/Place	-0.051900	...	0
ohio	-0.477144	...	0
state	-0.272753	...	0
place	0.230784	...	0
http://dbpedia.org/resource/Roman_Catholic_Dioc...	-0.452327	...	0
margate	-0.749552	...	0
city	0.080078	...	0
schools	-1.180198	...	0
stanford	-0.512758	...	0

This cluster corresponds to location data, as we can see the presence of postal codes, the ‘Place’ superclass, states and cities. It is likely that the subclusters correspond to four of the subclasses of ‘Place’ (address, postcode, city, state). As there is only one ‘country’ this would not be visible.

Cluster 1:



	0	...	label
http://www.city.ac.uk/ds/inm713/ollie_keers/mag...	-0.415374	...	1
http://www.city.ac.uk/ds/inm713/ollie_keers/Est...	0.311988	...	1
http://www.city.ac.uk/ds/inm713/ollie_keers/pas...	-0.354458	...	1
restaurant	0.186728	...	1
http://www.city.ac.uk/ds/inm713/ollie_keers/cas...	-0.723616	...	1
entertainment	0.305889	...	1
avocado	-0.106077	...	1
cafe	0.532168	...	1
side	-0.891771	...	1
of	0.128725	...	1

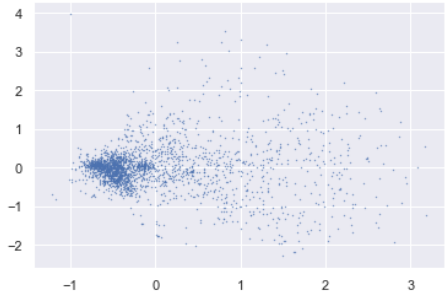
This cluster corresponds to establishment details & descriptors. ‘Avocado’ is present, indicating that some item descriptors are present in this cluster.

Cluster 2:

	0	...	label
product	-0.092805	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.381753	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.220769	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.127653	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.317476	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.402684	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.218028	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.273531	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	-0.176785	...	2
http://www.city.ac.uk/ds/inm713/ollie_keers/pro...	0.006360	...	2

This cluster corresponds to menu items, each of which has been created as product_X, where X is a unique number.

Cluster 3:



[10 rows x 101 columns]

	0	...	label
Type	0.307604	...	3
type	0.198887	...	3
http://www.city.ac.uk/ds/inm713/ollie_keers/piz...	-0.415116	...	3
food	-0.532233	...	3
http://www.city.ac.uk/ds/inm713/ollie_keers/Men...	-0.893749	...	3
cheesy	0.099829	...	3
item	-0.967977	...	3
pizza	0.295525	...	3
sauce	-0.809167	...	3
http://www.city.ac.uk/ds/inm713/ollie_keers/Food	-0.900414	...	3

This cluster corresponds to foods & food descriptors.
OWL2Vec* has separated the classes into distinct clusters. This has happened most successfully for the configuration with the most training. K-means clustering doesn’t appear to have successfully isolated each of the sub-classes, as can be seen in cluster zero. Future work could look at identifying the subclass sub-clusters either through reclustering each of the discovered class clusters, or employing a density based clustering technique such as OPTICS or DBSCAN.

Acknowledgement

Code provided by Ernesto Jimenez-Ruiz, as part of the Semantic Web Technologies & Knowledge Graphs module of the MSc Data Science course, has been used throughout this work. Where applicable, this has been referenced in comments in the code.