

task_rdf\pizza_kg.py

```

1  """
2  Some description
3  """
4  import re
5  import time
6
7  import pandas as pd
8  import numpy as np
9  from rdflib import Graph, Namespace, URIRef, Literal, RDFS
10 from rdflib.namespace import RDF, XSD
11 from rdflib.util import guess_format
12 import owlrl
13
14 import task_rdf.lookup as lookup
15 from task_rdf.isub import isub
16
17 # Default prefixes represent data from public knowledge graphs
18 DEFAULT_PREFIXES = [
19     ("dc", "http://purl.org/dc/elements/1.1/"),
20     ("owl", "http://www.w3.org/2002/07/owl#"),
21     ("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#"),
22     ("rdfs", "http://www.w3.org/2000/01/rdf-schema#"),
23     ("skos", "http://www.w3.org/2004/02/skos/core#"),
24     ("xml", "http://www.w3.org/XML/1998/namespace"),
25     ("xsd", "http://www.w3.org/2001/XMLSchema#"),
26     ("dbr", "http://dbpedia.org/resource/"),
27     ("dbo", "http://dbpedia.org/ontology/"),
28     ("wd", "http://www.wikidata.org/entity/"),
29     ("wdt", "http://www.wikidata.org/prop/direct/"),
30 ]
31
32
33 class PizzaKG(object):
34     # Setting of knowledge graphs object
35     file_path: str
36     namespace_str: str
37     prefix: str
38     entity_uri_dict: dict = {}
39     enable_external_uri: bool = True
40     external_uri_score_threshold = 0.4
41     noises = ["and", "/", "&", "."]
42     category_noise = ["Restaurant", "restaurant"]
43     meaningful_noise = {"and": " ", "or": " "}
44
45     def __init__(
46         self,
47         _file_path,
48         _name_space_str,
49         _name_space_prefix,
50         _prefixes=DEFAULT_PREFIXES,
51     ) -> None:
52         super().__init__()
53
54         # Setup input file as data
55         self.file_path = _file_path
56         self.data = pd.read_csv(

```

```

57         self.file_path,
58         delimiter=",",
59         quotechar='"',
60         escapechar="\\",
61         dtype={"postcode": str},
62     )
63
64     # Initialise the graph
65     self.graph = Graph()
66
67     # Setup customised name space
68     self.namespace_str = _name_space_str
69     self.namespace = Namespace(self.namespace_str)
70     self.graph.bind(_name_space_prefix, _name_space_str)
71
72     # Binding the prefixes
73     self.bind_prefixes(_prefixes)
74
75     # Initialise lookup service
76     self.dbpedia = lookup.DBpediaLookup()
77     self.wikidata = lookup.WikidataAPI()
78     self.google_kg = lookup.GoogleKGLookup()
79
80     # Preprocessing menu item
81     self.data["menu item"] = self.data["menu item"].apply(
82         lambda e: self.menu_name_preprocessing(e)
83     )
84
85     ##### MAIN TASK: CSV TO RDF CONVERSION #####
86     def convert_csv_to_rdf(self, _enable_external_uri: bool = enable_external_uri):
87         """
88         Convert data from dataframe to rdf
89         :param _enable_external_uri:
90         :return:
91         """
92
93         start_time = time.time()
94         print("##### STARTING CONVERSION #####")
95
96         # Country
97         self.graph.add(
98             (self.namespace.Country, RDFS.subClassOf, self.namespace.Location)
99         )
100        self.data["country"].apply(
101            lambda x: self.generate_type_triple(
102                entity=x,
103                class_type=self.namespace.Country,
104                _enable_external_uri=_enable_external_uri,
105                _category_filter="http://dbpedia.org/resource/Category:Lists_of_countries",
106            )
107        )
108        self.data.apply(
109            lambda row: self.generate_literal_triple(
110                entity=row["country"],
111                predicate=self.namespace.name,
112                literal=row["country"],
113                datatype=XSD.string,
114            ),
115            axis=1,

```

```
116         )
117
118     # State
119     self.graph.add((self.namespace.State, RDFS.subClassOf, self.namespace.Location))
120     self.data["state"].apply(
121         lambda x: self.generate_type_triple(
122             entity=x,
123             class_type=self.namespace.State,
124             _category_filter="http://dbpedia.org/resource/Category:States_of_the_United_States",
125             _external_uri_score_threshold=0.8,
126         )
127     )
128     self.data.apply(
129         lambda row: self.generate_literal_triple(
130             entity=row["state"],
131             predicate=self.namespace.name,
132             literal=row["state"],
133             datatype=XSD.string,
134         ),
135         axis=1,
136     )
137     self.data.apply(
138         lambda row: self.generate_object_triple(
139             subject=row["state"],
140             predicates=[
141                 self.namespace.isStateOf,
142                 self.namespace.locatedIn,
143                 self.namespace.locatedInCountry,
144             ],
145             object=row["country"],
146         ),
147         axis=1,
148     )
149
150     # City
151     self.graph.add((self.namespace.City, RDFS.subClassOf, self.namespace.Location))
152     self.data["city"].apply(
153         lambda x: self.generate_type_triple(
154             entity=x,
155             class_type=self.namespace.City,
156             _category_filter="http://dbpedia.org/resource/Category:Cities_in_the_United_States",
157             _external_uri_score_threshold=0.8,
158         )
159     )
160     self.data.apply(
161         lambda row: self.generate_literal_triple(
162             entity=row["city"],
163             predicate=self.namespace.name,
164             literal=row["city"],
165             datatype=XSD.string,
166         ),
167         axis=1,
168     )
169     self.data.apply(
170         lambda row: self.generate_object_triple(
171             subject=row["city"],
172             predicates=[self.namespace.locatedInState, self.namespace.locatedIn],
173             object=row["state"],
```

```

174         ),
175         axis=1,
176     )
177     self.data.apply(
178         lambda row: self.generate_object_triple(
179             subject=row["city"],
180             predicates=[self.namespace.locatedCountry, self.namespace.locatedIn],
181             object=row["country"],
182         ),
183         axis=1,
184     )
185
186     # Address
187     # We will have to concat a few field to make address identifier
188     self.graph.add(
189         (self.namespace.Address, RDFS.subClassOf, self.namespace.Location)
190     )
191     self.data["address_id"] = self.data[["address", "city", "state"]].apply(
192         lambda row: " ".join(row.astype(str)), axis=1
193     )
194     self.data["address_id"].apply(
195         lambda x: self.generate_type_triple(
196             entity=x, class_type=self.namespace.Address, _enable_external_uri=False
197         )
198     )
199     self.data.apply(
200         lambda row: self.generate_literal_triple(
201             entity=row["address_id"],
202             predicate=self.namespace.firstLineAddress,
203             literal=row["address"],
204             datatype=XSD.string,
205         ),
206         axis=1,
207     )
208     self.data.apply(
209         lambda row: self.generate_literal_triple(
210             entity=row["address_id"],
211             predicate=self.namespace.postCode,
212             literal=row["postcode"],
213             datatype=XSD.string,
214         ),
215         axis=1,
216     )
217     self.data.apply(
218         lambda row: self.generate_object_triple(
219             subject=row["address_id"],
220             predicates=[self.namespace.locatedCity, self.namespace.locatedIn],
221             object=row["city"],
222         ),
223         axis=1,
224     )
225     self.data.apply(
226         lambda row: self.generate_object_triple(
227             subject=row["address_id"],
228             predicates=[self.namespace.locatedState, self.namespace.locatedIn],
229             object=row["state"],
230         ),
231         axis=1,
232     )
233     self.data.apply(

```

```

234         lambda row: self.generate_object_triple(
235             subject=row["address_id"],
236             predicates=[self.namespace.locatedCountry, self.namespace.locatedIn],
237             object=row["country"],
238         ),
239         axis=1,
240     )
241
242     # Restaurant
243     self.graph.add(
244         (self.namespace.Restaurant, RDFS.subClassOf, self.namespace.Location)
245     )
246     self.data["restaurant_id"] = self.data[["name", "address_id"]].apply(
247         lambda row: " ".join(row.astype(str)), axis=1
248     )
249     self.data["restaurant_id"].apply(
250         lambda x: self.generate_type_triple(
251             entity=x,
252             class_type=self.namespace.Restaurant,
253             _enable_external_uri=False,
254         )
255     )
256     self.data.apply(
257         lambda row: self.generate_literal_triple(
258             entity=row["restaurant_id"],
259             predicate=self.namespace.name,
260             literal=row["name"],
261             datatype=XSD.string,
262         ),
263         axis=1,
264     )
265     self.data.apply(
266         lambda row: self.generate_object_triple(
267             subject=row["restaurant_id"],
268             predicates=[self.namespace.locatedAddress, self.namespace.locatedIn],
269             object=row["address_id"],
270         ),
271         axis=1,
272     )
273     self.data["categories"].apply(
274         lambda e: [
275             self.graph.add(
276                 (
277                     URIRef(self.generate_internal_class_name(_e)),
278                     RDFS.subClassOf,
279                     self.namespace.Restaurant,
280                 )
281             )
282             for _e in self.preprocessing_array_string(
283                 e, self.noises, self.category_noise
284             )
285         ]
286     )
287     self.data.apply(
288         lambda row: [
289             self.generate_type_triple(
290                 entity=row["restaurant_id"],
291                 class_type=(URIRef(self.generate_internal_class_name(_e))),
292                 _enable_external_uri=False,
293             )

```

```

294         for _e in self.preprocessing_array_string(
295             _input=row["categories"],
296             _noises=self.noises,
297             _element_noise=self.category_noise,
298         )
299     ],
300     axis=1,
301 )
302
303 # Currency
304 self.data["currency"].apply(
305     lambda x: self.generate_type_triple(
306         entity=x,
307         class_type=self.namespace.Currency,
308         _external_uri_score_threshold=0.8,
309     )
310 )
311 self.data.apply(
312     lambda row: self.generate_object_triple(
313         subject=row["currency"],
314         predicates=[self.namespace.currencyOfCountry],
315         object=row["country"],
316     ),
317     axis=1,
318 )
319 self.data.apply(
320     lambda row: self.generate_literal_triple(
321         entity=row["currency"],
322         predicate=self.namespace.name,
323         literal=row["currency"],
324         datatype=XSD.string,
325     ),
326     axis=1,
327 )
328
329 # Item value
330 # We will need to preprocess item value
331 self.data["item_value_id"] = self.data[["item value", "currency"]].apply(
332     lambda row: "".join(row.astype(str))
333     if (
334         self.is_missing(row["item value"]) == False
335         and self.is_missing(row["currency"]) == False
336     )
337     else "",
338     axis=1,
339 )
340 self.data["item_value_id"].apply(
341     lambda x: self.generate_type_triple(
342         entity=x,
343         class_type=self.namespace.ItemValue,
344         _enable_external_uri=False,
345     )
346 )
347 self.data.apply(
348     lambda row: self.generate_object_triple(
349         subject=row["item_value_id"],
350         predicates=[self.namespace.amountCurrency],
351         object=row["currency"],
352     ),
353     axis=1,

```

```

354     )
355     self.data.apply(
356         lambda row: self.generate_literal_triple(
357             entity=row["item_value_id"],
358             predicate=self.namespace.amount,
359             literal=row["item value"],
360             datatype=XSD.double,
361         ),
362         axis=1,
363     )
364
365     # Pizza & Ingredient
366     self.graph.add((self.namespace.MenuItem, RDFS.subClassOf, self.namespace.Food))
367     self.graph.add((self.namespace.Ingredient, RDFS.subClassOf, self.namespace.Food))
368     self.data["item_id"] = self.data[["menu item", "restaurant_id"]].apply(
369         lambda row: " ".join(row.astype(str)), axis=1
370     )
371     self.data["item_id"].apply(
372         lambda x: self.generate_type_triple(
373             entity=x,
374             class_type=self.namespace.MenuItem,
375             _external_uri_score_threshold=0.7,
376         )
377     )
378     self.data.apply(
379         lambda row: self.generate_literal_triple(
380             entity=row["item_id"],
381             predicate=self.namespace.name,
382             literal=row["menu item"],
383             datatype=XSD.string,
384         ),
385         axis=1,
386     )
387     self.data.apply(
388         lambda row: self.generate_object_triple(
389             subject=row["item_id"],
390             predicates=[self.namespace.servedInRestaurant],
391             object=row["restaurant_id"],
392         ),
393         axis=1,
394     )
395     self.data.apply(
396         lambda row: self.generate_object_triple(
397             subject=row["item_id"],
398             predicates=[self.namespace.hasValue],
399             object=row["item_value_id"],
400         ),
401         axis=1,
402     )
403
404     # Item description
405     self.data.apply(
406         lambda row: self.generate_description(
407             row, _enable_external_uri=self.enable_external_uri
408         ),
409         axis=1,
410     )
411
412     print(
413         "##### CONVERSION FINISHED IN: {} SECONDS #####".format(

```

```

414         time.time() - start_time
415     )
416 )
417
418 ##### PREPROCESSINGS #####
419 def bind_prefixes(self, prefixes):
420     """
421     Since we will link the data with public KG, we will need to define
422     and bind the prefixes for them
423     :param prefixes:
424     :return: None
425     """
426     for prefix in prefixes:
427         self.graph.bind(prefix[0], prefix[1])
428
429 def menu_name_preprocessing(self, item_name: str):
430     """
431     We do notice that there are some pizza name that in this format:
432     "Pizza, Margherita", we will try to find and match them using
433     regular expression and then change them to format "margherita pizza"
434     :param item_name:
435     :return: processed item name
436     """
437
438     # Menu item pattern, for example "Pizza, Margherita"
439     pattern = re.compile(r"^pizza\s?,\s?[a-z\s]+.$")
440
441     # Match result
442     matched = re.search(pattern, item_name.lower())
443
444     # Check if match exist and then replace
445     if matched:
446         return re.sub(r"(\w+), (\w+)", r"\2 \1", item_name)
447     return item_name
448
449 def process_entity_lexical(self, _entity: str):
450     """
451     Remove characters that could break URI
452     :param _entity:
453     :return:
454     """
455     pattern = re.compile("[\W_]+")
456     return pattern.sub("_", _entity)
457
458 def preprocessing_array_string(
459     self,
460     _input: str,
461     _noises: [str],
462     _element_noise: [str] = [],
463     _meaningful_noise={},
464 ):
465     for noise, replacement in _meaningful_noise.items():
466         _input = _input.replace(noise, replacement)
467     for noise in _noises:
468         _input = _input.replace(noise, "")
469     _input = re.sub(" +", " ", _input).strip()
470     _input_arr = _input.split(",")
471     _input_arr = [e.rstrip("s").strip() for e in _input_arr]
472     _input_arr = [e for e in _input_arr if e not in _element_noise]
473     return _input_arr

```



```

474
475 ##### ENTITY GENERATION #####
476 def generate_uri(
477     self,
478     entity: str,
479     _enable_external_uri: bool = enable_external_uri,
480     _category_filter: str = "",
481     _external_uri_score_threshold: float = external_uri_score_threshold,
482 ):
483     """
484     We will generate URI for the entity, note that there are some logic:
485     - We can choose if we need to use external KG
486     - We can also choose the threshold for lexical similiarity
487     For example: if the lexical similarity is too low, we would rather create
entity URI
488     in our default namespace
489     :param entity:
490     :param _enable_external_uri:
491     :param _category_filter:
492     :return:
493     """
494     uri = self.namespace_str + self.process_entity_lexical(_entity=entity)
495
496     if _enable_external_uri:
497         _uri, _score = self.generate_external_uri(
498             _query=entity,
499             _category_filter=_category_filter,
500         )
501         if (_uri != "") & (_score >= _external_uri_score_threshold):
502             uri = _uri
503
504     self.entity_uri_dict[entity.lower()] = uri
505
506     return uri
507
508 def generate_external_uri(
509     self, _query: str, _category_filter: str = "", _limit: int = 5
510 ):
511     """
512     Use pre-written lookup code to look for the enitivity on services.
513     Currently, we are implementing DBpedia and Wikidata.
514     We will expect to extend the search to Google KG in the future
515     since entity URI from Google KG is different from other services
516     :param _query:
517     :param _category_filter:
518     :param _limit:
519     :return: uri:
520     """
521
522     # Query all services and return
523     dbpedia_result = self.dbpedia.getKGEntities(
524         query=_query, limit=_limit, category_filter=_category_filter
525     )
526     # wikidata_result = self.wikidata.getKGEntities(query=_query, limit=_limit)
527
528     # Mute wikidata if we specify search category inside DBpedia
529     if re.search(r"dbpedia\.org", _category_filter):
530         wikidata_result = []
531
532     # Concentrate the result and then return

```

```

533     entities = [
534         *dbpedia_result,
535         #         *wikidata_result,
536     ]
537
538     # Parameters for comparation
539     score = -1
540     uri = ""
541
542     # Iterate returned array and check for the most correct result
543     if entities:
544         for entity in entities:
545             _score = isub(_query, entity.label)
546             if _score > score:
547                 uri = entity.ident
548                 score = _score
549
550     return uri, score
551
552 def generate_internal_class_name(self, _name: str):
553     pattern = re.compile("[\W_]+")
554     return self.namespace_str + pattern.sub("", _name).capitalize()
555
556 ##### TRIPLES GENERATIONS #####
557 def generate_type_triple(
558     self,
559     entity: str,
560     class_type: str,
561     _enable_external_uri: bool = enable_external_uri,
562     _category_filter: str = "",
563     _external_uri_score_threshold: float = external_uri_score_threshold,
564 ):
565     """
566     Generate type triple: Example: ns:London rdf:type ns:City
567     :param entity:
568     :param class_type:
569     :param _enable_external_uri:
570     :param _category_filter:
571     :return:
572     """
573     # Check blank or empty
574     if self.is_missing(entity):
575         return
576
577     # Check if item exist in dictionary so that we don't have to call API again
578     if entity.lower() in self.entity_uri_dict:
579         uri = self.entity_uri_dict[entity.lower()]
580     else:
581         uri = self.generate_uri(
582             entity=entity,
583             _enable_external_uri=_enable_external_uri,
584             _category_filter=_category_filter,
585             _external_uri_score_threshold=_external_uri_score_threshold,
586         )
587
588     # Add type triple
589     self.graph.add((URIRef(uri), RDF.type, class_type))
590
591 def generate_literal_triple(
592     self, entity: str, predicate: str, literal: str, datatype: str

```

```

593     ):
594         """
595         Generate literal triple: Example: ns:London ns:name "London"^^xsd:string
596         :param entity:
597         :param predicate:
598         :param literal:
599         :param datatype:
600         :return:
601         """
602         # If the literal is blank or empty, we will pass it
603         if self.is_missing(literal):
604             return
605
606         # Get the URI from dictionary
607         uri = self.entity_uri_dict[entity.lower()]
608
609         # Get the literal
610         _literal = Literal(literal, datatype=datatype)
611
612         # Add literal to graph
613         self.graph.add((URIRef(uri), predicate, _literal))
614
615     def generate_object_triple(self, subject: str, predicates: [str], object: str):
616         """
617         Generate literal triple: Example: ns:USD ns:isCurrencyOf ns:US
618         :param subject:
619         :param predicate:
620         :param object:
621         :return:
622         """
623         # If the literal is blank or empty, we will pass it
624         if self.is_missing(subject) or self.is_missing(object):
625             return
626
627         # Get the URI from dictionary
628         subject_uri = self.entity_uri_dict[subject.lower()]
629         object_uri = self.entity_uri_dict[object.lower()]
630
631         for predicate in predicates:
632             self.graph.add((URIRef(subject_uri), predicate, URIRef(object_uri)))
633
634     def generate_description(
635         self, row, _enable_external_uri: bool = enable_external_uri
636     ):
637         if not _enable_external_uri:
638             self.generate_literal_triple(
639                 entity=row["item_id"],
640                 predicate=self.namespace.description,
641                 literal=row["item description"],
642                 datatype=XSD.string,
643             )
644         else:
645             if self.is_missing(row["item description"]):
646                 pass
647             else:
648                 desc_array = self.preprocessing_array_string(
649                     _input=row["item description"],
650                     _noises=self.noises,
651                     _meaningful_noise=self.meaningful_noise,
652                 )

```

```

653         for desc in desc_array:
654             if self.is_missing(desc):
655                 pass
656             else:
657                 uri = self.generate_uri(
658                     entity=desc,
659                     _enable_external_uri=_enable_external_uri,
660
661 _category_filter="https://dbpedia.org/page/Category:Food_ingredients",
662                     _external_uri_score_threshold=0.65,
663                 )
664                 if uri.startswith(self.namespace):
665                     self.generate_literal_triple(
666                         entity=row["item_id"],
667                         predicate=self.namespace.description,
668                         literal=desc,
669                         datatype=XSD.string,
670                     )
671                 else:
672 self.graph.add((URIRef(uri), RDFS.subClassOf,
673 self.namespace.Ingredient))
674                 ingredient_id = desc + "_" + row["item_id"]
675                 self.generate_type_triple(
676                     entity=ingredient_id,
677                     class_type=URIRef(uri),
678                     _enable_external_uri=False
679                 )
680                 self.generate_object_triple(
681                     subject=ingredient_id,
682                     predicates=[self.namespace.isIngredientOf],
683                     object=row["item_id"],
684                 )
685
686 ##### REASONING #####
687 def perform_reasoning(self, ontology: str):
688     """
689     Perform reasoning with existing ontology
690     :param ontology:
691     :return:
692     """
693     # Load the ontology file
694     self.graph.load(ontology, format=guess_format(ontology))
695
696     # Load and expand reasoner
697     owlrl.DeductiveClosure(
698         owlrl.OWLRL.OWLRL_Semantics,
699         axiomatic_triples=False,
700         datatype_axioms=False,
701     ).expand(self.graph)
702
703     print("Done reasoning, triples count: '" + str(len(self.graph)) + "'.")
704
705 ##### SAVE GRAPH #####
706 def save_graph(self, output_file: str, _format: str = "ttl"):
707     """
708     Just a function to save graph as file
709     :param output_file:
710     :param _format:
711     :return:
712     """

```

```
711         self.graph.serialize(destination=output_file, format=_format)
712
713     ##### VALIDATIONS #####
714     def is_missing(self, value: str):
715         """
716         Check if a value is empty, blank or missing
717         :param value:
718         :return:
719         """
720         return (
721             (value != value)
722             or (value is None)
723             or (value == "")
724             or (value == " ")
725             or (value == np.nan)
726             or (value == "_")
727             or (value == "nan")
728         )
729
```

task_vector\graph_combination.py

```
1  from rdflib import Graph
2  from rdflib.util import guess_format
3
4
5  class GraphCombination(object):
6      graph_paths: [str]
7      output_path: str
8      graph: Graph
9
10     def __init__(self, _graph_paths: [str], _output_path: str) -> None:
11         super().__init__()
12         self.graph_paths = _graph_paths
13         self.output_path = _output_path
14         self.graph = Graph()
15
16     def combination_and_save(self, _output_format: str = "ttl"):
17         for _e in self.graph_paths:
18             self.graph.load(_e, format=guess_format(_e))
19
20         self.graph.serialize(destination=self.output_path, format=_output_format)
21         return self.output_path
22
```

task_vector\helper.py

```
1 from matplotlib import pyplot as plt
2 from gensim.models import KeyedVectors
3 from sklearn.decomposition import PCA
4
5
6 def visualise_2d(p, labels, _target_terms, circle_size: float = 2.0, _fig_size=(15, 10)):
7     plt.figure(figsize=_fig_size)
8     plt.scatter(p[:, 0], p[:, 1], c="lightcoral")
9     fig = plt.gcf()
10    ax = fig.gca()
11    ax.set_aspect("equal")
12
13    for x, y, label in zip(p[:, 0], p[:, 1], labels):
14        if label in _target_terms:
15            target_zone = plt.Circle((x, y), circle_size, color="mediumturquoise",
fill=False)
16            ax.add_patch(target_zone)
17            plt.annotate(label, xy=(x + 0.05, y + 0.05), color="orangered")
18        else:
19            plt.annotate(label, xy=(x + 0.05, y + 0.05), color="black")
20
21
22 def dim_reduction(
23     _model: KeyedVectors, _target_terms: [str], _topn: int = 20, _n_dim: int = 2
24 ):
25     model, similar_terms = generate_word_model(_model, _target_terms, _topn)
26     pca = PCA(n_components=_n_dim)
27     P = pca.fit_transform(model)
28     labels = similar_terms
29     return P, labels
30
31
32 def generate_word_model(_model: KeyedVectors, _target_terms: [str], _topn: int = 20):
33     target_dict = {
34         term: list(
35             map(
36                 lambda x: x[0],
37                 _model.wv.most_similar_cosmul(positive=term.split(" "), topn=_topn),
38             )
39         )
40         for term in _target_terms
41     }
42     similar_terms = (
43         sum(map(lambda x: [x[0]] + x[1], target_dict.items()), []) + _target_terms
44     )
45     return _model.wv[similar_terms], similar_terms
46
```

In [1]:

```
%load_ext autoreload
%autoreload 2
```

...

In [29]:

```
from graph_combination import GraphCombination
from gensim.models import KeyedVectors
import helper
```

Ontology Embeddings (Task Vector)

Subtask Vector.1

Graphs concentration

In [3]:

```
# We will perform graphs concentration
graph_paths = [
    "../cw_onto/pizza-restaurants-ontology.ttl",
    "../task_rdf/pizza_restaurant.ttl"
]

output_path = "concentrated_graph.owl"

concentrated_graph = GraphCombination(_graph_paths=graph_paths, _output_path=output_path)
print("Concentrated graph created: {}".format(concentrated_graph.combination_and_save(_ou
```

Concentrated graph created: concentrated_graph.owl

Run OWL2Vec

```
cd OWL2Vec-Star-master/ owl2vec_star standalone --config_file
../owl2vec_config/config_1.cfg owl2vec_star standalone --config_file
../owl2vec_config/config_2.cfg owl2vec_star standalone --config_file
../owl2vec_config/config_3.cfg
```

The configuration was modified from default configuration

config_1.cfg

```
iteration = 15
```

config_2.cfg

```
axiom_reasoner = hermit iteration = 20
```

config_3.cfg


```
axiom_reasoner = elk walk_depth = 5 iteration = 30
```

File savings

Binary and textual files saved to owl2vec_embeddings

Subtask Vector.2

In [4]:

```
# First, we need to load output file from OWL2Vec
model = KeyedVectors.load("./owl2vec_embeddings/config_2_output/ontology.embeddings", mmap_mode='r')
wv = model.wv
```

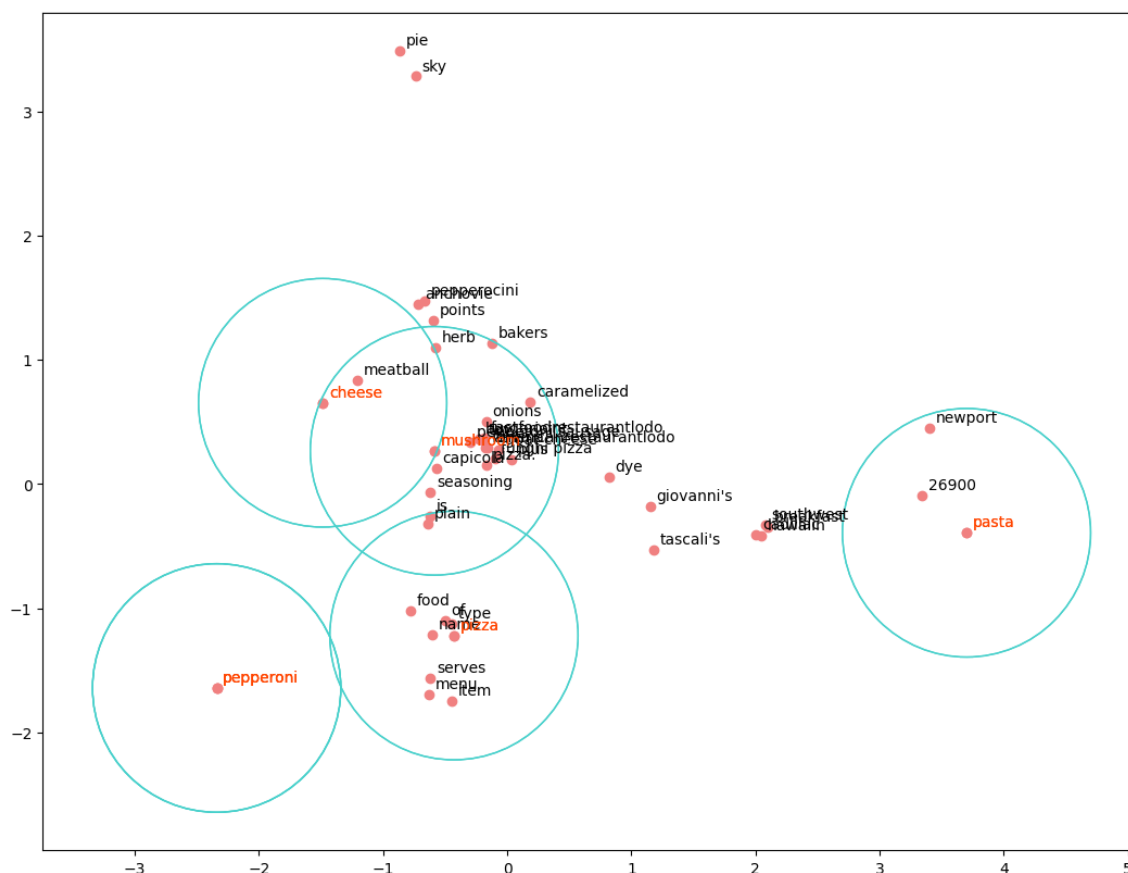
3 pairs with high similarity

In [91]:

```
# We will compare similarity of
# "pizza" and "pepperoni"
term_pair_0 = ["pizza", "pepperoni"]
term_pair_1 = ["pasta", "pepperoni"]
term_pair_2 = ["mushroom", "cheese"]

term_pair = term_pair_0 + term_pair_1 + term_pair_2

model_2d, label = helper.dim_reduction(_model=model, _target_terms=term_pair, _topn=8, _radius=1.5)
helper.visualise_2d(p=model_2d, labels=label, circle_size=1, target_terms=term_pair)
```



We can see that the "pizza" has a close similarity to "pepperoni", "mushroom" and "cheese" since all of them is ingredient in pizza while pasta has a high dissimilarity

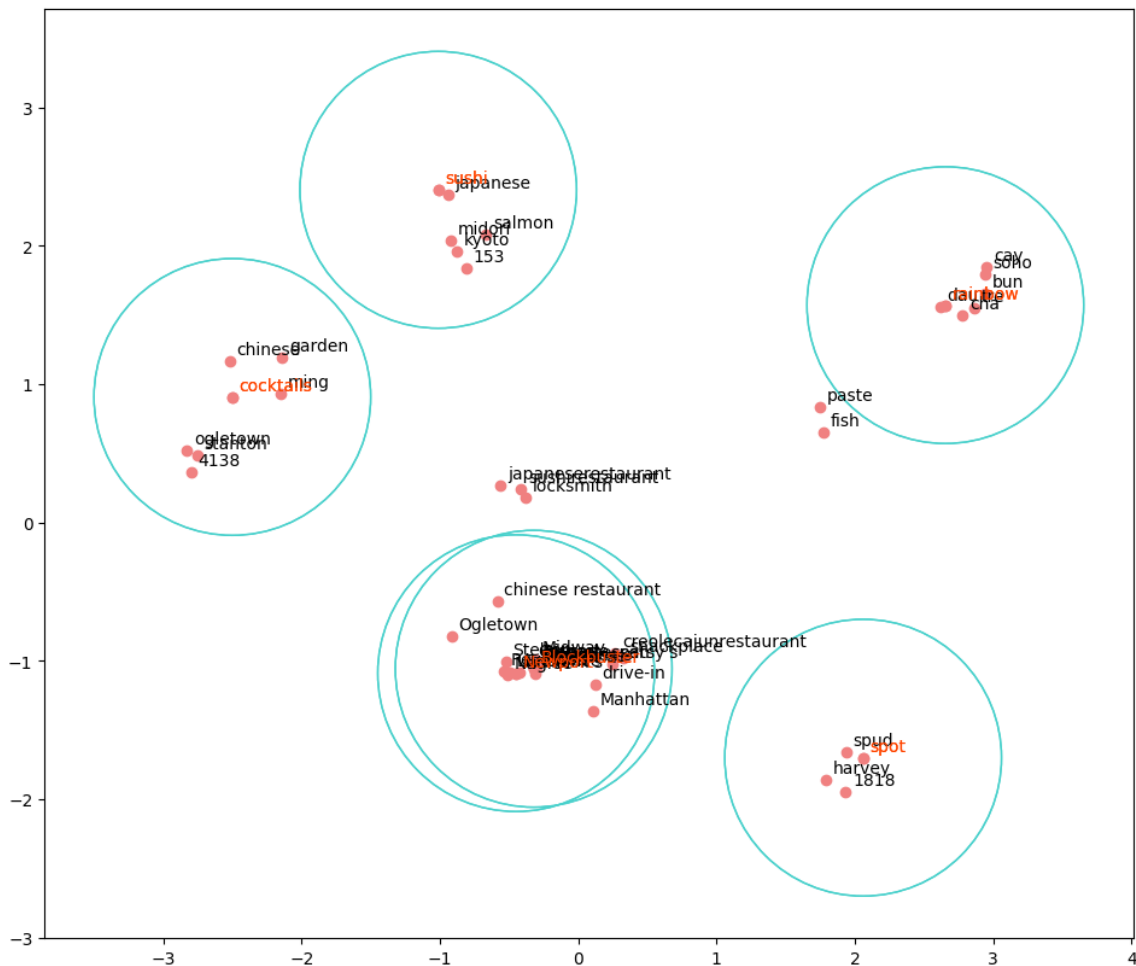
3 pairs with high dissimilarity

In [114]:

```
# We will compare similarity of
# "ingredient", "rainbow"
# "ingredient", "rainbow"
# "cocktails", "spot"
term_pair_0 = ["Blockbuster", "Newport"]
term_pair_1 = ["sushi", "rainbow"]
term_pair_2 = ["cocktails", "spot"]

term_pair = term_pair_0 + term_pair_1 + term_pair_2

model_2d, label = helper.dim_reduction(_model=model, _target_terms=term_pair, _topn=8, _r=
helper.visualise_2d(p=model_2d, labels=label, circle_size=1, _target_terms=term_pair)
```



As shown in the graph: "rainbow", "sushi", "cocktails", "spot" have decent dissimilarity while "Blockbuster" and "Newport" are close since they might be the location name

task_alignment\task_alignment.py

```
1 from rdflib import Graph, Namespace, URIRef, OWL, RDFS
2 from rdflib.plugins.sparql import prepareQuery
3 import Levenshtein as lev
4
5
6 # Task OA1
7 def find_equivalences(onto1, onto2):
8     # Defining the owl namespace to be added to the turtle file
9     owl = Namespace("http://www.w3.org/2002/07/owl#")
10    owl_equivalentClass = owl.equivalentClass
11
12    # Iteration of the entities in the cw_onto file
13    for entity1 in onto1.subjects():
14        name1 = onto1.value(
15            subject=entity1,
16            predicate=URIRef("http://www.w3.org/2000/01/rdf-schema#label"),
17        )
18
19    # Iteration of the entities in the pizza.owl file
20    for entity2 in onto2.subjects():
21        name2 = onto2.value(
22            subject=entity2,
23            predicate=URIRef("http://www.w3.org/2000/01/rdf-schema#label"),
24        )
25
26    # Computation of jaro-winkler similarity to compare both inputted strings
27    similarity = lev.jaro_winkler(name1, name2)
28
29    # check if similarity is high enough to be considered equivalent
30    if similarity > 0.9:
31        # add triple to the alignment_graph
32        equivalence_triple = (entity1, owl_equivalentClass, entity2)
33        alignment_graph.add(equivalence_triple)
34
35
36 # TASK OA2
37 # The function combines all sources in a single graph
38 def create_graph_with_reasoning():
39     single_combined_graph = Graph()
40     # TASK OA2.1
41     single_combined_graph += cw_onto_graph
42     # TASK OA2.2
43     single_combined_graph += pizza_graph
44     # TASK OA2.3
45     single_combined_graph += alignment_graph
46     # TASK OA2.4
47     single_combined_graph += cw_data_graph
48     # Reasoning being applied
49     single_combined_graph.bind("owl", OWL)
50     single_combined_graph.bind("rdfs", RDFS)
51     single_combined_graph.bind(
52         "cw", "http://www.semanticweb.org/city/in3067-inm713/2023/restaurants#"
53     )
54
55     # Turtle file is created of the combined graph
```

```
56     single_combined_graph.serialize("./task_alignment/alignment_results/combined_task.ttl",
format="turtle")
57
58
59 def query_pizza(onto):
60     pizza = Namespace("http://www.co-ode.org/ontologies/pizza/pizza.owl#")
61
62     onto.bind("pizza", pizza)
63
64     query_string = """
65         PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
66         SELECT * WHERE {
67             ?topping rdfs:subClassOf pizza:PizzaTopping .
68         }
69     """
70     query = prepareQuery(query_string)
71     results = onto.query(query)
72
73
74     results.serialize(destination="./test.csv", format='csv')
75
76
77 pizza_loc = "./pizza_ontology/pizza.owl"
78 cw_onto_loc = "./cw_onto/pizza-restaurants-ontology.owl"
79 cw_data_loc = "./task_rdf/pizza_restaurant.ttl"
80
81 pizza_graph = Graph()
82 pizza_graph.parse(pizza_loc, format="xml")
83
84 cw_onto_graph = Graph()
85 cw_onto_graph.parse(cw_onto_loc, format="xml")
86
87 cw_data_graph = Graph()
88 cw_data_graph.parse(cw_data_loc, format="ttl")
89
90 #query_pizza(pizza_graph)
91
92 alignment_graph = Graph()
93
94 find_equivalences(cw_onto_graph, pizza_graph)
95
96 alignment_graph.serialize("./task_alignment/alignment_results/equivalences.ttl",
format="turtle")
97
98 create_graph_with_reasoning()
99
```

task_sparql\sparql_task.py

```

1  from rdflib import Graph, Namespace
2  from rdflib.plugins.sparql import prepareQuery
3
4
5  class SparqlQuery:
6      def __init__(self, filename):
7          ttl_file = ".ttl"
8          rdf_file = ".rdf"
9          self.graph = Graph()
10         if filename.find(ttl_file):
11             self.graph.parse(filename, format="turtle")
12         elif filename.find(rdf_file):
13             self.graph.parse(filename, format="xml")
14         else:
15             print("File type given was incorrect. Needs to be in .ttl or .rdf format")
16
17     # Creates the query and saves the result into a csv file.
18     def make_query_to_csv(self, query, output_file):
19         result = self.graph.query(query)
20         print(f"Result: {result} Length: {len(result)}")
21         for row in result:
22             print(row)
23         result.serialize(destination=f"{output_file}_results.csv", format="csv")
24
25
26  if __name__ == "__main__":
27     filename = "./task_rdf/pizza_restaurant.ttl"
28
29     sparql_query = SparqlQuery(filename)
30
31     # This query retrieves all the restaurants in the state of Texas
32     def task1():
33         query = """
34         SELECT ?name ?firstLineAddress ?cityName ?stateName
35         WHERE {
36             ?restaurant cw:name ?name .
37             ?restaurant a cw:Restaurant .
38             ?restaurant cw:locatedAddress ?address .
39             ?address cw:firstLineAddress ?firstLineAddress .
40             ?address cw:locatedCity ?city .
41             ?city cw:name ?cityName .
42             ?address cw:locatedState ?state .
43             ?state cw:name ?stateName .
44             FILTER (?stateName = "TX")
45         }
46         """
47         sparql_query.make_query_to_csv(query,
48             "./task_sparql/sparql_result/SPARQL1_subtask")
49
50     # This query returns the average price of all items on the Burgers & Cupcakes menu
51     def task2():
52         query = """
53         SELECT ?restaurantName (AVG(?menu_item_price) AS ?avg_value)
54         WHERE {
55             ?menuItem cw:servedInRestaurant ?restaurant .
56             ?menuItem cw:hasValue ?value .

```

```

56         ?value cw:amount ?menu_item_price .
57         ?restaurant a cw:Restaurant .
58         ?restaurant cw:name ?restaurantName .
59         FILTER (?restaurantName = "Burgers & Cupcakes")
60     }
61     """
62     sparql_query.make_query_to_csv(query,
63     "./task_sparql/sparql_result/SPARQL2_subtask")
64
65     # This query returns the number of restaurants in all the cities except the ones in
66     the
67     # state of Washington
68     def task3():
69         query = """
70         SELECT ?cityName (COUNT(?restaurant) AS ?num_restaurants)
71         WHERE {
72             ?restaurant cw:locatedAddress ?address .
73             ?address cw:firstLineAddress ?firstLineAddress .
74             ?address cw:locatedCity ?city .
75             ?address cw:locatedState ?state .
76             ?state cw:name ?stateName .
77             ?city cw:name ?cityName .
78             FILTER (?stateName != "WA")
79         }
80         GROUP BY ?cityName
81         HAVING (COUNT(?restaurant))
82         """
83         sparql_query.make_query_to_csv(query,
84         "./task_sparql/sparql_result/SPARQL3_subtask")
85
86         # This query returns the cities with a number of restaurants higher than 7 AND
87         # an average item price of more than 10
88         def task4():
89             query = """
90             SELECT ?cityName (COUNT(?restaurant) AS ?num_restaurants) (AVG(?menu_item_price)
91             AS ?avg_value)
92             WHERE {
93                 ?restaurant cw:locatedAddress ?address .
94                 ?address cw:firstLineAddress ?firstLineAddress .
95                 ?address cw:locatedCity ?city .
96                 ?city cw:name ?cityName .
97                 ?menuItem cw:servedInRestaurant ?restaurant .
98                 ?menuItem cw:hasValue ?value .
99                 ?value cw:amount ?menu_item_price .
100             }
101             GROUP BY ?cityName
102             HAVING (COUNT(?restaurant) > 7 && AVG(?menu_item_price) > 10)
103             ORDER BY DESC(?num_restaurants) ?avg_value
104             """
105             sparql_query.make_query_to_csv(query,
106             "./task_sparql/sparql_result/SPARQL4_subtask")
107
108             # This query returns the names of the restaurants that are either in New York City
109             # or don't have any items on the menu worth higher than 5 USD
110             def task5():
111                 query = """
112                 SELECT ?name ?cityName ?menu_item_price
113                 WHERE {
114                     {
115                         ?restaurant cw:name ?name .
116                         ?restaurant a cw:Restaurant .

```

```
112         ?restaurant cw:locatedAddress ?address .
113         ?address cw:firstLineAddress ?firstLineAddress .
114         ?address cw:locatedCity ?city .
115         ?city cw:name "New York".
116     } UNION {
117         ?restaurant cw:name ?name .
118         ?restaurant a cw:Restaurant .
119         ?restaurant cw:locatedAddress ?address .
120         ?address cw:firstLineAddress ?firstLineAddress .
121         ?address cw:locatedCity ?city .
122         ?menuItem cw:servedInRestaurant ?restaurant .
123         ?menuItem cw:hasValue ?value .
124         ?value cw:amount ?menu_item_price .
125         ?city cw:name ?cityName .
126         FILTER NOT EXISTS {
127             ?menuItem cw:servedInRestaurant ?restaurant .
128             ?menuItem cw:hasValue ?value .
129             ?value cw:amount ?menu_item_price .
130             FILTER (?menu_item_price > 5)
131         }
132     }
133 }
134 """
135     sparql_query.make_query_to_csv(query,
136     "./task_sparql/sparql_result/SPARQL5_subtask")
137
138     task5()
```


task_rdf\tabular_to_kg.py

```

1  from task_rdf.pizza_kg import PizzaKG
2
3
4  # Constrains and settings
5  FILE_PATH = "../cw_data/IN3067-INM713_coursework_data_pizza_500.csv"
6  NAMESPACE_STR = "http://www.semanticweb.org/city/in3067-inm713/2023/restaurants#"
7  NAMESPACE_PREFIX = "cw"
8  ONTOLOGY = "../cw_onto/pizza-restaurants-ontology.ttl"
9
10
11 if __name__ == "__main__":
12     pizza_kg = PizzaKG(
13         _file_path=FILE_PATH,
14         _name_space_str=NAMESPACE_STR,
15         _name_space_prefix=NAMESPACE_PREFIX,
16     )
17
18     """
19     TASK RDF.2: RDF generation
20     In this task, we will perform the creating RDFs from CSV file without external public
    KG vocabulary usage.
21     In order to do this, please check `pizza_kg.py` and change line:
22     `enable_external_uri: bool = False`
23     """
24     pizza_kg.convert_csv_to_rdf()
25     pizza_kg.save_graph("pizza_restaurant_offline.ttl")
26
27     """
28     TASK RDF.3: Perform reasoning with `cw_onto`
29     We will run the reasoning with offline file
30     """
31     pizza_kg.perform_reasoning(ontology=ONTOLOGY)
32     pizza_kg.save_graph("pizza_restaurant_offline_reasoned.ttl")
33
34     """
35     TASK RDF.4: Reuse URIs from state-of-the art knowledge graphs
36     To run this task, please put TASK RDF.2 & TASK RDF.3 in comment.
37     And check `pizza_kg.py` and change line:
38     `enable_external_uri: bool = False`
39     """
40     pizza_kg.convert_csv_to_rdf()
41     pizza_kg.save_graph("pizza_restaurant.ttl")
42     pizza_kg.perform_reasoning(ontology=ONTOLOGY)
43     pizza_kg.save_graph("pizza_restaurant_reasoned.ttl")
44
45     """
46     TASK RDF.5: Exploit an external Knowledge Graph to perform disambiguation
47     We already exploit external KGs to perform logical tasks, however, we didn't choose
48     to correct location name.
49     As the `description` column of this data is mixed between normal pizza description
50     and some ingredient, we use external KG to identify what is the description and
51     what is the ingredient. With external KG disabled, ingredient data in `description`
52     columns won't be classified.
53     We will discuss this further in report.
54     """
55

```