task_rdf\pizza_kg.py

```python
"""
Some description
"""
import re
import time

import pandas as pd
import numpy as np
from rdflib import Graph, Namespace, URIRef, Literal, RDFS
from rdflib.namespace import RDF, XSD
from rdflib.util import guess_format
import owlrl

import task_rdf.lookup as lookup
from task_rdf.isub import isub

# Default prefixes represent data from public knowledge graphs
DEFAULT_PREFIXES = [
    ("dc", "http://purl.org/dc/elements/1.1/"),
    ("owl", "http://www.w3.org/2002/07/owl#"),
    ("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#"),
    ("rdfs", "http://www.w3.org/2000/01/rdf-schema#"),
    ("skos", "http://www.w3.org/2004/02/skos/core#"),
    ("xml", "http://www.w3.org/XML/1998/namespace"),
    ("xsd", "http://www.w3.org/2001/XMLSchema#"),
    ("dbr", "http://dbpedia.org/resource/"),
    ("dbo", "http://dbpedia.org/ontology/"),
    ("wd", "http://www.wikidata.org/entity/"),
    ("wdt", "http://www.wikidata.org/prop/direct/"),
]


class PizzaKG(object):
    # Setting of knowledge graphs object
    file_path: str
    namespace_str: str
    prefix: str
    entity_uri_dict: dict = {}
    enable_external_uri: bool = True
    external_uri_score_threshold = 0.4
    noises = ["and", "/", "&", "."]
    category_noise = ["Restaurant", "restaurant"]
    meaningful_noise = {"and": ",", "or": ","}

    def __init__(
        self,
        _file_path,
        _name_space_str,
        _name_space_prefix,
        _prefixes=DEFAULT_PREFIXES,
    ) -> None:
        super().__init__()

        # Setup input file as data
        self.file_path = _file_path
        self.data = pd.read_csv(
```

```
            self.file_path,
            delimiter=",",
            quotechar='"',
            escapechar="\\",
            dtype={"postcode": str},
        )

        # Initialise the graph
        self.graph = Graph()

        # Setup customised name space
        self.namespace_str = _name_space_str
        self.namespace = Namespace(self.namespace_str)
        self.graph.bind(_name_space_prefix, _name_space_str)

        # Binding the prefixes
        self.bind_prefixes(_prefixes)

        # Initialise lookup service
        self.dbpedia = lookup.DBpediaLookup()
        self.wikidata = lookup.WikidataAPI()
        self.google_kg = lookup.GoogleKGLookup()

        # Preprocessing menu item
        self.data["menu item"] = self.data["menu item"].apply(
            lambda e: self.menu_name_preprocessing(e)
        )

    ######### MAIN TASK: CSV TO RDF CONVERSION #########
    def convert_csv_to_rdf(self, _enable_external_uri: bool = enable_external_uri):
        """
        Convert data from dataframe to rdf
        :param _enable_external_uri:
        :return:
        """

        start_time = time.time()
        print("######### STARTING CONVERSION #########")

        # Country
        self.graph.add(
            (self.namespace.Country, RDFS.subClassOf, self.namespace.Location)
        )
        self.data["country"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.Country,
                _enable_external_uri=_enable_external_uri,
_category_filter="http://dbpedia.org/resource/Category:Lists_of_countries",
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["country"],
                predicate=self.namespace.name,
                literal=row["country"],
                datatype=XSD.string,
            ),
            axis=1,
```

```
        )

        # State
        self.graph.add((self.namespace.State, RDFS.subClassOf, self.namespace.Location))
        self.data["state"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.State,
                _category_filter="http://dbpedia.org/resource/Category:States_of_the_United_States",
                _external_uri_score_threshold=0.8,
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["state"],
                predicate=self.namespace.name,
                literal=row["state"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["state"],
                predicates=[
                    self.namespace.isStateOf,
                    self.namespace.locatedIn,
                    self.namespace.locatedInCountry,
                ],
                object=row["country"],
            ),
            axis=1,
        )

        # City
        self.graph.add((self.namespace.City, RDFS.subClassOf, self.namespace.Location))
        self.data["city"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.City,
                _category_filter="http://dbpedia.org/resource/Category:Cities_in_the_United_States",
                _external_uri_score_threshold=0.8,
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["city"],
                predicate=self.namespace.name,
                literal=row["city"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["city"],
                predicates=[self.namespace.locatedInState, self.namespace.locatedIn],
                object=row["state"],
```

```
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["city"],
                predicates=[self.namespace.locatedCountry, self.namespace.locatedIn],
                object=row["country"],
            ),
            axis=1,
        )

        # Address
        # We will have to concat a few field to make address identifier
        self.graph.add(
            (self.namespace.Address, RDFS.subClassOf, self.namespace.Location)
        )
        self.data["address_id"] = self.data[["address", "city", "state"]].apply(
            lambda row: " ".join(row.astype(str)), axis=1
        )
        self.data["address_id"].apply(
            lambda x: self.generate_type_triple(
                entity=x, class_type=self.namespace.Address, _enable_external_uri=False
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["address_id"],
                predicate=self.namespace.firstLineAddress,
                literal=row["address"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["address_id"],
                predicate=self.namespace.postCode,
                literal=row["postcode"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["address_id"],
                predicates=[self.namespace.locatedCity, self.namespace.locatedIn],
                object=row["city"],
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["address_id"],
                predicates=[self.namespace.locatedState, self.namespace.locatedIn],
                object=row["state"],
            ),
            axis=1,
        )
        self.data.apply(
```

```
            lambda row: self.generate_object_triple(
                subject=row["address_id"],
                predicates=[self.namespace.locatedCountry, self.namespace.locatedIn],
                object=row["country"],
            ),
            axis=1,
        )

        # Restaurant
        self.graph.add(
            (self.namespace.Restaurant, RDFS.subClassOf, self.namespace.Location)
        )
        self.data["restaurant_id"] = self.data[["name", "address_id"]].apply(
            lambda row: " ".join(row.astype(str)), axis=1
        )
        self.data["restaurant_id"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.Restaurant,
                _enable_external_uri=False,
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["restaurant_id"],
                predicate=self.namespace.name,
                literal=row["name"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["restaurant_id"],
                predicates=[self.namespace.locatedAddress, self.namespace.locatedIn],
                object=row["address_id"],
            ),
            axis=1,
        )
        self.data["categories"].apply(
            lambda e: [
                self.graph.add(
                    (
                        URIRef(self.generate_internal_class_name(_e)),
                        RDFS.subClassOf,
                        self.namespace.Restaurant,
                    )
                )
                for _e in self.preprocessing_array_string(
                    e, self.noises, self.category_noise
                )
            ]
        )
        self.data.apply(
            lambda row: [
                self.generate_type_triple(
                    entity=row["restaurant_id"],
                    class_type=(URIRef(self.generate_internal_class_name(_e))),
                    _enable_external_uri=False,
                )
```

```
                for _e in self.preprocessing_array_string(
                    _input=row["categories"],
                    _noises=self.noises,
                    _element_noise=self.category_noise,
                )
            ],
            axis=1,
        )

        # Currency
        self.data["currency"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.Currency,
                _external_uri_score_threshold=0.8,
            )
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["currency"],
                predicates=[self.namespace.currencyOfCountry],
                object=row["country"],
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["currency"],
                predicate=self.namespace.name,
                literal=row["currency"],
                datatype=XSD.string,
            ),
            axis=1,
        )

        # Item value
        # We will need to preprocess item value
        self.data["item_value_id"] = self.data[["item value", "currency"]].apply(
            lambda row: "".join(row.astype(str))
            if (
                self.is_missing(row["item value"]) == False
                and self.is_missing(row["currency"]) == False
            )
            else "",
            axis=1,
        )
        self.data["item_value_id"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.ItemValue,
                _enable_external_uri=False,
            )
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["item_value_id"],
                predicates=[self.namespace.amountCurrency],
                object=row["currency"],
            ),
            axis=1,
```

```
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["item_value_id"],
                predicate=self.namespace.amount,
                literal=row["item value"],
                datatype=XSD.double,
            ),
            axis=1,
        )

        # Pizza & Ingredient
        self.graph.add((self.namespace.MenuItem, RDFS.subClassOf, self.namespace.Food))
        self.graph.add((self.namespace.Ingredient, RDFS.subClassOf, self.namespace.Food))
        self.data["item_id"] = self.data[["menu item", "restaurant_id"]].apply(
            lambda row: " ".join(row.astype(str)), axis=1
        )
        self.data["item_id"].apply(
            lambda x: self.generate_type_triple(
                entity=x,
                class_type=self.namespace.MenuItem,
                _external_uri_score_threshold=0.7,
            )
        )
        self.data.apply(
            lambda row: self.generate_literal_triple(
                entity=row["item_id"],
                predicate=self.namespace.name,
                literal=row["menu item"],
                datatype=XSD.string,
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["item_id"],
                predicates=[self.namespace.servedInRestaurant],
                object=row["restaurant_id"],
            ),
            axis=1,
        )
        self.data.apply(
            lambda row: self.generate_object_triple(
                subject=row["item_id"],
                predicates=[self.namespace.hasValue],
                object=row["item_value_id"],
            ),
            axis=1,
        )

        # Item description
        self.data.apply(
            lambda row: self.generate_description(
                row, _enable_external_uri=self.enable_external_uri
            ),
            axis=1,
        )

        print(
            "######### CONVERSION FINISHED IN: {} SECONDS #########".format(
```

```
                time.time() - start_time
            )
        )

    ######### PREPROCESSINGS #########
    def bind_prefixes(self, prefixes):
        """
        Since we will link the data with public KG, we will need to define
        and bind the prefixes for them
        :param prefixes:
        :return: None
        """
        for prefix in prefixes:
            self.graph.bind(prefix[0], prefix[1])

    def menu_name_preprocessing(self, item_name: str):
        """
        We do notice that there are some pizza name that in this format:
        "Pizza, Margherita", we will try to find and match them using
        regular expression and then change them to format "margherita pizza"
        :param item_name:
        :return: processed item name
        """

        # Menu item pattern, for example "Pizza, Margherita"
        pattern = re.compile(r"^pizza\s?,\s?[a-z\s]+.$")

        # Match result
        matched = re.search(pattern, item_name.lower())

        # Check if match exist and then replace
        if matched:
            return re.sub(r"(\w+), (\w+)", r"\2 \1", item_name)
        return item_name

    def process_entity_lexical(self, _entity: str):
        """
        Remove characters that could break URI
        :param _entity:
        :return:
        """
        pattern = re.compile("[\W_]+")
        return pattern.sub("_", _entity)

    def preprocessing_array_string(
        self,
        _input: str,
        _noises: [str],
        _element_noise: [str] = [],
        _meaningful_noise={},
    ):
        for noise, replacement in _meaningful_noise.items():
            _input = _input.replace(noise, replacement)
        for noise in _noises:
            _input = _input.replace(noise, "")
        _input = re.sub(" +", " ", _input).strip()
        _input_arr = _input.split(",")
        _input_arr = [e.rstrip("s").strip() for e in _input_arr]
        _input_arr = [e for e in _input_arr if e not in _element_noise]
        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 enitity 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
```

```
    ):
        """
        Generate literal triple: Example: ns:London ns:name "London"^^xsd:string
        :param entity:
        :param predicate:
        :param literal:
        :param datatype:
        :return:
        """
        # If the literal is blank or empty, we will pass it
        if self.is_missing(literal):
            return

        # Get the URI from dictionary
        uri = self.entity_uri_dict[entity.lower()]

        # Get the literal
        _literal = Literal(literal, datatype=datatype)

        # Add literal to graph
        self.graph.add((URIRef(uri), predicate, _literal))

    def generate_object_triple(self, subject: str, predicates: [str], object: str):
        """
        Generate literal triple: Example: ns:USD ns:isCurrencyOf ns:US
        :param subject:
        :param predicate:
        :param object:
        :return:
        """
        # If the literal is blank or empty, we will pass it
        if self.is_missing(subject) or self.is_missing(object):
            return

        # Get the URI from dictionary
        subject_uri = self.entity_uri_dict[subject.lower()]
        object_uri = self.entity_uri_dict[object.lower()]

        for predicate in predicates:
            self.graph.add((URIRef(subject_uri), predicate, URIRef(object_uri)))

    def generate_description(
        self, row, _enable_external_uri: bool = enable_external_uri
    ):
        if not _enable_external_uri:
            self.generate_literal_triple(
                entity=row["item_id"],
                predicate=self.namespace.description,
                literal=row["item description"],
                datatype=XSD.string,
            )
        else:
            if self.is_missing(row["item description"]):
                pass
            else:
                desc_array = self.preprocessing_array_string(
                    _input=row["item description"],
                    _noises=self.noises,
                    _meaningful_noise=self.meaningful_noise,
                )
```

```
                for desc in desc_array:
                    if self.is_missing(desc):
                        pass
                    else:
                        uri = self.generate_uri(
                            entity=desc,
                            _enable_external_uri=_enable_external_uri,
_category_filter="https://dbpedia.org/page/Category:Food_ingredients",
                            _external_uri_score_threshold=0.65,
                        )
                        if uri.startswith(self.namespace):
                            self.generate_literal_triple(
                                entity=row["item_id"],
                                predicate=self.namespace.description,
                                literal=desc,
                                datatype=XSD.string,
                            )
                        else:
                            self.graph.add((URIRef(uri), RDFS.subClassOf,
self.namespace.Ingredient))
                            ingredient_id = desc + "_" + row["item_id"]
                            self.generate_type_triple(
                                entity=ingredient_id,
                                class_type=URIRef(uri),
                                _enable_external_uri=False
                            )
                            self.generate_object_triple(
                                subject=ingredient_id,
                                predicates=[self.namespace.isIngredientOf],
                                object=row["item_id"],
                            )

    ######### REASONING #########
    def perform_reasoning(self, ontology: str):
        """
        Perform reasoning with existing ontology
        :param ontology:
        :return:
        """
        # Load the ontology file
        self.graph.load(ontology, format=guess_format(ontology))

        # Load and expand reasoner
        owlrl.DeductiveClosure(
            owlrl.OWLRL.OWLRL_Semantics,
            axiomatic_triples=False,
            datatype_axioms=False,
        ).expand(self.graph)

        print("Done reasoning, triples count: '" + str(len(self.graph)) + "'.")

    ######### SAVE GRAPH #########
    def save_graph(self, output_file: str, _format: str = "ttl"):
        """
        Just a function to save graph as file
        :param output_file:
        :param _format:
        :return:
        """
```

```
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

```python
from rdflib import Graph
from rdflib.util import guess_format


class GraphCombination(object):
    graph_paths: [str]
    output_path: str
    graph: Graph

    def __init__(self, _graph_paths: [str], _output_path: str) -> None:
        super().__init__()
        self.graph_paths = _graph_paths
        self.output_path = _output_path
        self.graph = Graph()

    def combination_and_save(self, _output_format: str = "ttl"):
        for _e in self.graph_paths:
            self.graph.load(_e, format=guess_format(_e))

        self.graph.serialize(destination=self.output_path, format=_output_format)
        return self.output_path

```

task_vector\helper.py

```python
from matplotlib import pyplot as plt
from gensim.models import KeyedVectors
from sklearn.decomposition import PCA


def visualise_2d(p, labels, _target_terms, circle_size: float = 2.0, _fig_size=(15, 10)):
    plt.figure(figsize=_fig_size)
    plt.scatter(p[:, 0], p[:, 1], c="lightcoral")
    fig = plt.gcf()
    ax = fig.gca()
    ax.set_aspect("equal")

    for x, y, label in zip(p[:, 0], p[:, 1], labels):
        if label in _target_terms:
            target_zone = plt.Circle((x, y), circle_size, color="mediumturquoise", fill=False)
            ax.add_patch(target_zone)
            plt.annotate(label, xy=(x + 0.05, y + 0.05), color="orangered")
        else:
            plt.annotate(label, xy=(x + 0.05, y + 0.05), color="black")


def dim_reduction(
    _model: KeyedVectors, _target_terms: [str], _topn: int = 20, _n_dim: int = 2
):
    model, similar_terms = generate_word_model(_model, _target_terms, _topn)
    pca = PCA(n_components=_n_dim)
    P = pca.fit_transform(model)
    labels = similar_terms
    return P, labels


def generate_word_model(_model: KeyedVectors, _target_terms: [str], _topn: int = 20):
    target_dict = {
        term: list(
            map(
                lambda x: x[0],
                _model.wv.most_similar_cosmul(positive=term.split(" "), topn=_topn),
            )
        )
        for term in _target_terms
    }
    similar_terms = (
        sum(map(lambda x: [x[0]] + x[1], target_dict.items()), []) + _target_terms
    )
    return _model.wv[similar_terms], similar_terms

```

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", mma
wv = model.wv
```

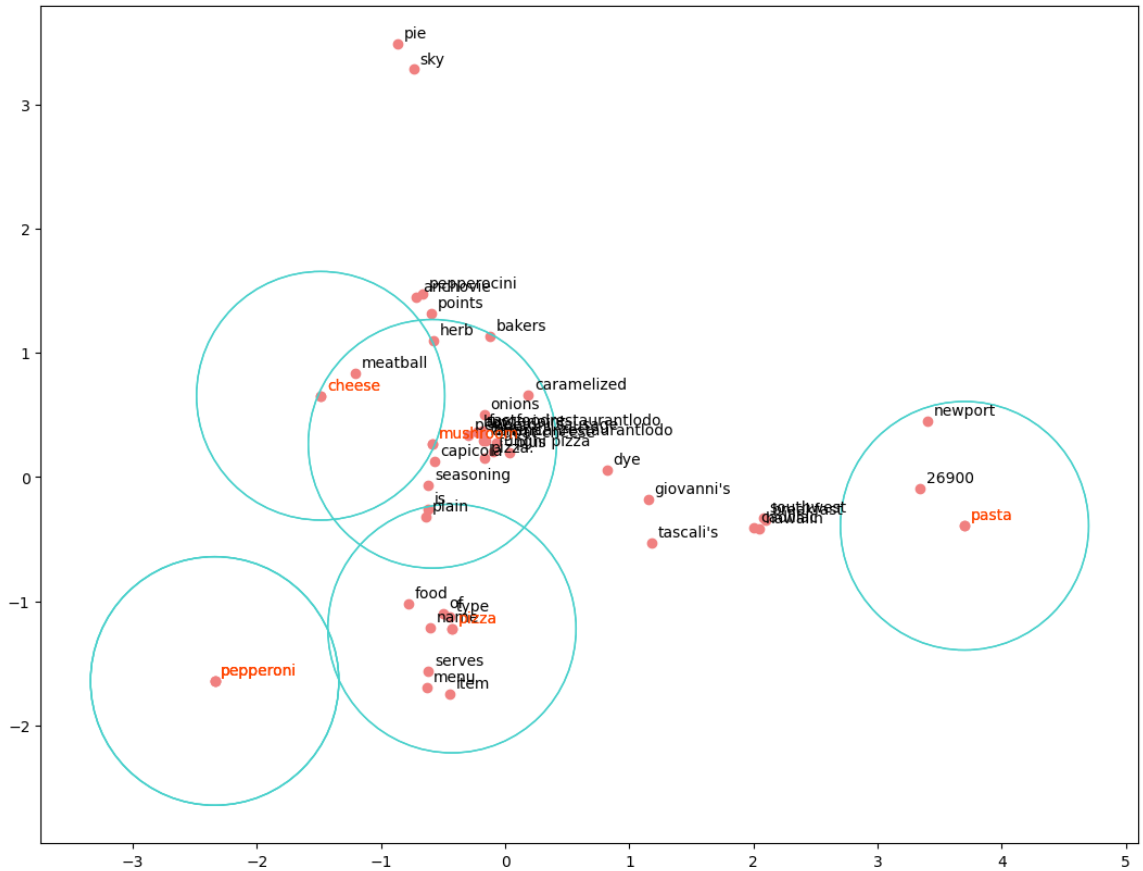
3 pairs with high similarity

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, _r
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

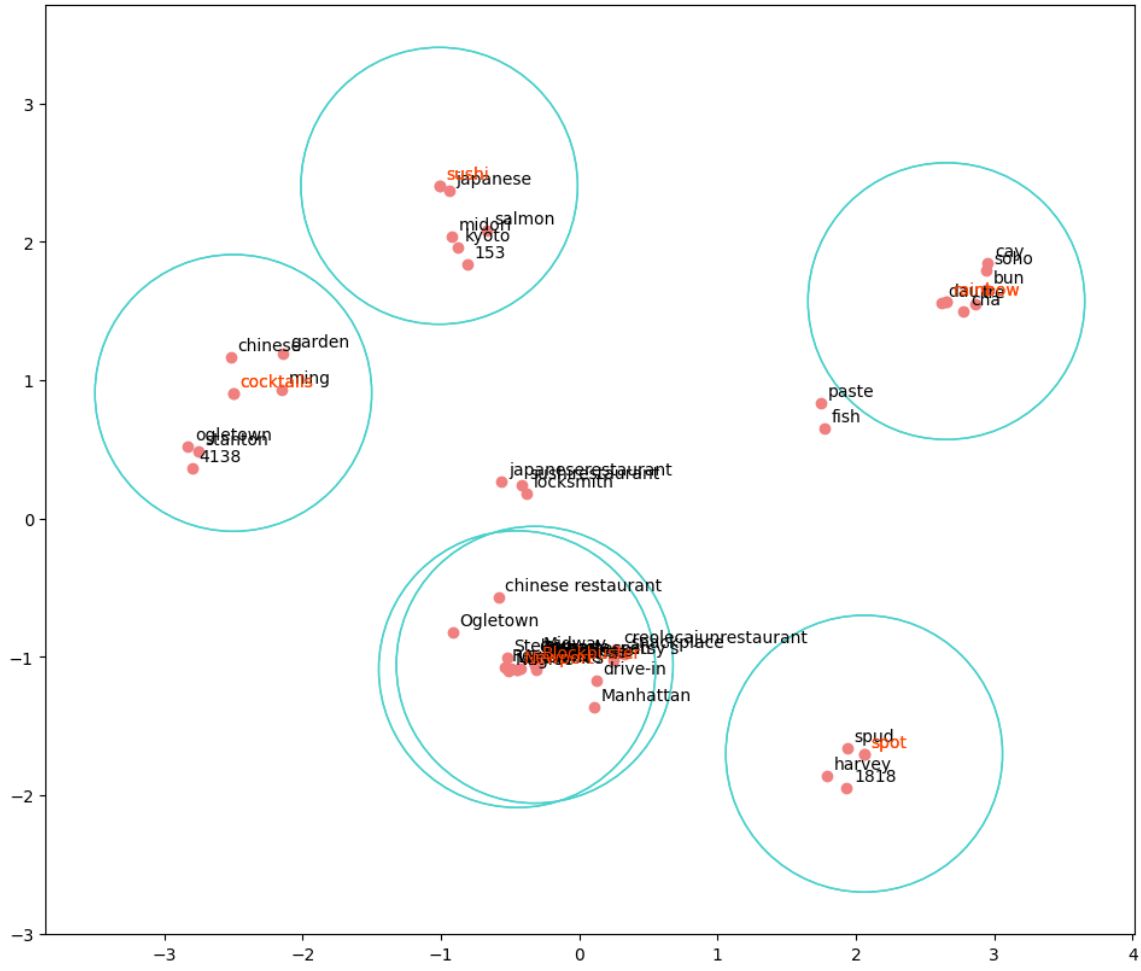
3 pairs with high dissimilarity

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

```python
from rdflib import Graph, Namespace, URIRef, OWL, RDFS
from rdflib.plugins.sparql import prepareQuery
import Levenshtein as lev


# Task OA1
def find_equivalences(onto1, onto2):
    # Defining the owl namespace to be added to the turtle file
    owl = Namespace("http://www.w3.org/2002/07/owl#")
    owl_equivalentClass = owl.equivalentClass

    # Iteration of the entities in the cw_onto file
    for entity1 in onto1.subjects():
        name1 = onto1.value(
            subject=entity1,
            predicate=URIRef("http://www.w3.org/2000/01/rdf-schema#label"),
        )

        # Iteration of the entities in the pizza.owl file
        for entity2 in onto2.subjects():
            name2 = onto2.value(
                subject=entity2,
                predicate=URIRef("http://www.w3.org/2000/01/rdf-schema#label"),
            )

            # Computation of jaro-winkler similarity to compare both inputed strings
            similarity = lev.jaro_winkler(name1, name2)

            # check if similarity is high enough to be considered equivalent
            if similarity > 0.9:
                # add triple to the alignment_graph
                equivalence_triple = (entity1, owl_equivalentClass, entity2)
                alignment_graph.add(equivalence_triple)


# TASK OA2
# The function combines all sources in a single graph
def create_graph_with_reasoning():
    single_combined_graph = Graph()
    # TASK OA2.1
    single_combined_graph += cw_onto_graph
    # TASK OA2.2
    single_combined_graph += pizza_graph
    # TASK OA2.3
    single_combined_graph += alignment_graph
    # TASK OA2.4
    single_combined_graph += cw_data_graph
    # Reasoning being applied
    single_combined_graph.bind("owl", OWL)
    single_combined_graph.bind("rdfs", RDFS)
    single_combined_graph.bind(
        "cw", "http://www.semanticweb.org/city/in3067-inm713/2023/restaurants#"
    )

    # Turtle file is created of the combined graph
```

```
    single_combined_graph.serialize("./task_alignment/alignment_results/combined_task.ttl", format="turtle")


def query_pizza(onto):
    pizza = Namespace("http://www.co-ode.org/ontologies/pizza/pizza.owl#")

    onto.bind("pizza", pizza)

    query_string = """
        PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
        SELECT * WHERE {
            ?topping rdfs:subClassOf pizza:PizzaTopping .
        }
    """
    query = prepareQuery(query_string)
    results = onto.query(query)


    results.serialize(destination="./test.csv",format='csv')


pizza_loc = "./pizza_ontology/pizza.owl"
cw_onto_loc = "./cw_onto/pizza-restaurants-ontology.owl"
cw_data_loc = "./task_rdf/pizza_restaurant.ttl"

pizza_graph = Graph()
pizza_graph.parse(pizza_loc, format="xml")

cw_onto_graph = Graph()
cw_onto_graph.parse(cw_onto_loc, format="xml")

cw_data_graph = Graph()
cw_data_graph.parse(cw_data_loc, format="ttl")

#query_pizza(pizza_graph)

alignment_graph = Graph()

find_equivalences(cw_onto_graph, pizza_graph)

alignment_graph.serialize("./task_alignment/alignment_results/equivalences.ttl", format="turtle")

create_graph_with_reasoning()

```

task_sparql\sparql_task.py

```python
from rdflib import Graph, Namespace
from rdflib.plugins.sparql import prepareQuery


class SparqlQuery:
    def __init__(self, filename):
        ttl_file = ".ttl"
        rdf_file = ".rdf"
        self.graph = Graph()
        if filename.find(ttl_file):
            self.graph.parse(filename, format="turtle")
        elif filename.find(rdf_file):
            self.graph.parse(filename, format="xml")
        else:
            print("File type given was incorrect. Needs to be in .ttl or .rdf format")

    # Creates the query and saves the result into a csv file.
    def make_query_to_csv(self, query, output_file):
        result = self.graph.query(query)
        print(f"Result: {result} Length: {len(result)}")
        for row in result:
            print(row)
        result.serialize(destination=f"{output_file}_results.csv", format="csv")


if __name__ == "__main__":
    filename = "./task_rdf/pizza_restaurant.ttl"

    sparql_query = SparqlQuery(filename)

    # This query retrieves all the restaurants in the state of Texas
    def task1():
        query = """
        SELECT ?name ?firstLineAddress ?cityName  ?stateName
        WHERE {
            ?restaurant cw:name ?name .
            ?restaurant a cw:Restaurant .
            ?restaurant cw:locatedAddress ?address .
            ?address cw:firstLineAddress ?firstLineAddress .
            ?address cw:locatedCity ?city .
            ?city cw:name ?cityName .
            ?address cw:locatedState ?state .
            ?state cw:name ?stateName .
            FILTER (?stateName = "TX")
        }
        """
        sparql_query.make_query_to_csv(query, "./task_sparql/sparql_result/SPARQL1_subtask")

    # This query returns the average price of all items on the Burgers & Cupcakes menu
    def task2():
        query = """
        SELECT ?restaurantName (AVG(?menu_item_price) AS ?avg_value)
        WHERE {
            ?menuItem cw:servedInRestaurant ?restaurant .
            ?menuItem cw:hasValue ?value .
```

```
            ?value cw:amount ?menu_item_price .
            ?restaurant a cw:Restaurant .
            ?restaurant cw:name ?restaurantName .
            FILTER (?restaurantName = "Burgers & Cupcakes")
        }
        """
        sparql_query.make_query_to_csv(query, "./task_sparql/sparql_result/SPARQL2_subtask")

    # This query returns the number of restaurants in all the cities except the ones in the
    # state of Washington
    def task3():
        query = """
        SELECT ?cityName (COUNT(?restaurant) AS ?num_restaurants)
        WHERE {
                ?restaurant cw:locatedAddress ?address .
                ?address cw:firstLineAddress ?firstLineAddress .
                ?address cw:locatedCity ?city .
                ?address cw:locatedState ?state .
                ?state cw:name ?stateName .
                ?city cw:name ?cityName .
                FILTER (?stateName != "WA")
        }
        GROUP BY ?cityName
        HAVING (COUNT(?restaurant))
        """
        sparql_query.make_query_to_csv(query, "./task_sparql/sparql_result/SPARQL3_subtask")

    # This query returns the cities with a number of restaurants higher than 7 AND
    # an average item price of more than 10
    def task4():
        query = """
        SELECT ?cityName (COUNT(?restaurant) AS ?num_restaurants) (AVG(?menu_item_price) AS ?avg_value)
        WHERE {
            ?restaurant cw:locatedAddress ?address .
            ?address cw:firstLineAddress ?firstLineAddress .
            ?address cw:locatedCity ?city .
            ?city cw:name ?cityName .
            ?menuItem cw:servedInRestaurant ?restaurant .
            ?menuItem cw:hasValue ?value .
            ?value cw:amount ?menu_item_price .
        }
        GROUP BY ?cityName
        HAVING (COUNT(?restaurant) > 7 && AVG(?menu_item_price) > 10)
        ORDER BY DESC(?num_restaurants) ?avg_value
        """
        sparql_query.make_query_to_csv(query, "./task_sparql/sparql_result/SPARQL4_subtask")

    # This query 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 USD
    def task5():
        query = """
        SELECT ?name ?cityName ?menu_item_price
        WHERE {
            {
                ?restaurant cw:name ?name .
                ?restaurant a cw:Restaurant .
```

```
                ?restaurant cw:locatedAddress ?address .
                ?address cw:firstLineAddress ?firstLineAddress .
                ?address cw:locatedCity ?city .
                ?city cw:name "New York".
            } UNION {
                ?restaurant cw:name ?name .
                ?restaurant a cw:Restaurant .
                ?restaurant cw:locatedAddress ?address .
                ?address cw:firstLineAddress ?firstLineAddress .
                ?address cw:locatedCity ?city .
                ?menuItem cw:servedInRestaurant ?restaurant .
                ?menuItem cw:hasValue ?value .
                ?value cw:amount ?menu_item_price .
                ?city cw:name ?cityName .
                FILTER NOT EXISTS {
                    ?menuItem cw:servedInRestaurant ?restaurant .
                    ?menuItem cw:hasValue ?value .
                    ?value cw:amount ?menu_item_price .
                    FILTER (?menu_item_price > 5)
                }
            }
        }
        """
        sparql_query.make_query_to_csv(query, "./task_sparql/sparql_result/SPARQL5_subtask")

    task5()

```

task_rdf\tabular_to_kg.py

```python
from task_rdf.pizza_kg import PizzaKG


# Constrains and settings
FILE_PATH = "../cw_data/IN3067-INM713_coursework_data_pizza_500.csv"
NAMESPACE_STR = "http://www.semanticweb.org/city/in3067-inm713/2023/restaurants#"
NAMESPACE_PREFIX = "cw"
ONTOLOGY = "../cw_onto/pizza-restaurants-ontology.ttl"


if __name__ == "__main__":
    pizza_kg = PizzaKG(
        _file_path=FILE_PATH,
        _name_space_str=NAMESPACE_STR,
        _name_space_prefix=NAMESPACE_PREFIX,
    )

    """
    TASK RDF.2: RDF generation
    In this task, we will perform the creating RDFs from CSV file without external public KG vocabulary usage.
    In order to do this, please check `pizza_kg.py` and change line:
    `enable_external_uri: bool = False`
    """
    pizza_kg.convert_csv_to_rdf()
    pizza_kg.save_graph("pizza_restaurant_offline.ttl")

    """
    TASK RDF.3: Perform reasoning with `cw_onto`
    We will run the reasoning with offline file
    """
    pizza_kg.perform_reasoning(ontology=ONTOLOGY)
    pizza_kg.save_graph("pizza_restaurant_offline_reasoned.ttl")

    """
    TASK RDF.4: Reuse URIs from state-of-the art knowledge graphs
    To run this task, please put TASK RDF.2 & TASK RDF.3 in comment.
    And check `pizza_kg.py` and change line:
    `enable_external_uri: bool = False`
    """
    pizza_kg.convert_csv_to_rdf()
    pizza_kg.save_graph("pizza_restaurant.ttl")
    pizza_kg.perform_reasoning(ontology=ONTOLOGY)
    pizza_kg.save_graph("pizza_restaurant_reasoned.ttl")

    """
    TASK RDF.5: Exploit an external Knowledge Graph to perform disambiguation
    We already exploit external KGs to perform logical taskes, however, we didn't choose
    to correct location name.
    As the `description` column of this data is mixed between normal pizza description
    and some ingredient, we use external KG to identify what is the description and
    what is the ingredient. With external KG disabled, ingredient data in `description`
    columns won't be classified.
    We will discuss this further in report.
    """

```