

**INM713 - *Semantic Web Technologies and Knowledge Graphs*  
Coursework Project**

Zacharias Detorakis

[zacharias.detorakis@city.ac.uk](mailto:zacharias.detorakis@city.ac.uk)

**Abstract**

This report aims at describing the steps taken and assumptions made as part of the coursework for the Semantic Web Technologies & Knowledge Graphs module. The goal of the coursework is to create a basic ontology that can fit the data from a csv file and generate the triples into a Knowledge Graph (KG). Following that there are tasks for reasoning, aligning the created ontology with an existing pizza ontology and querying the results triples to get insights from the underlying data. As a final task the OWL2Vec tool is used to create embeddings and used to create clusters of the ontology entities.

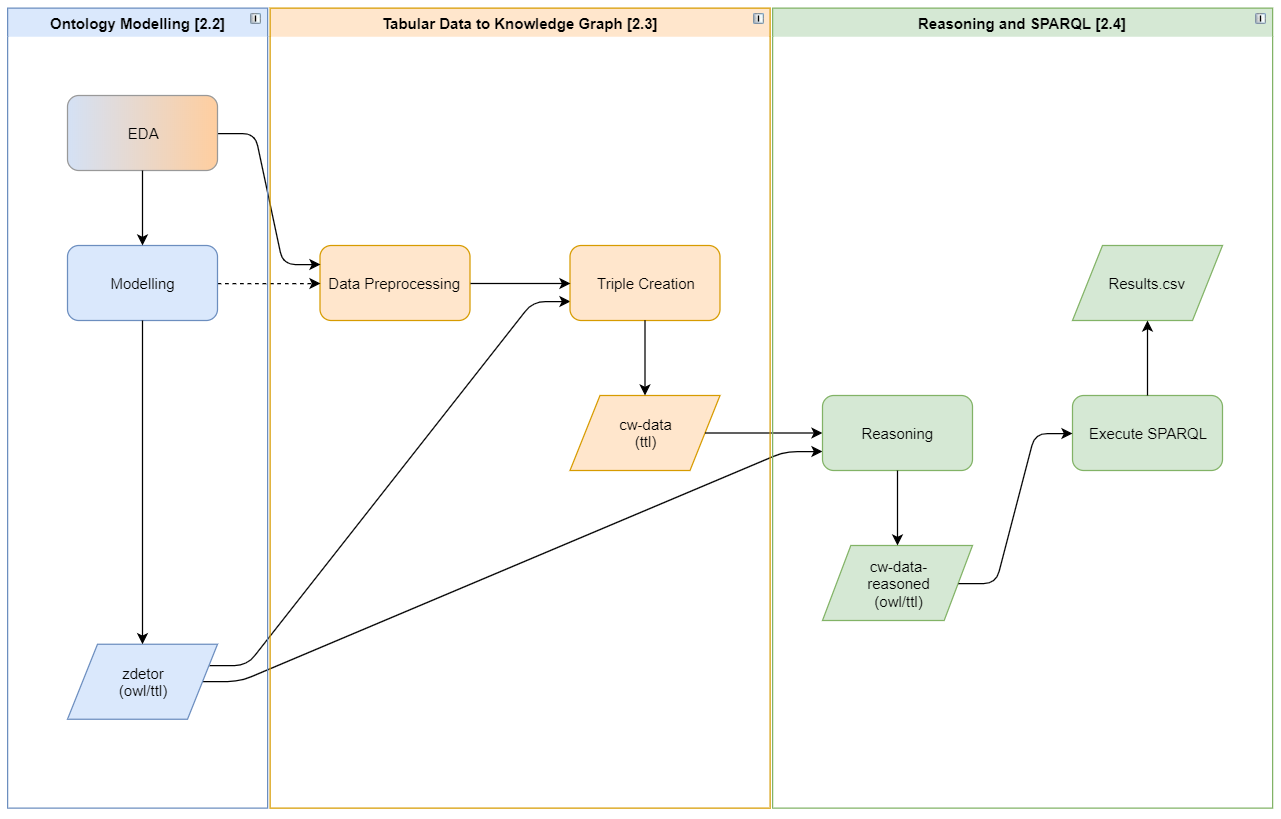


Figure 1. Process of modelling and converting tabular data to a KG

# Introduction

The following diagram [Figure 1] illustrates the steps taken for the first 3 tasks of the coursework (i.e. to create an ontology based of the give data, load the data to a KG and extract useful insights).

# Ontology Modelling [2.2]

Prior to jumping straight into modelling the first step was to understand the nature of the data we are trying to follow, and data discrepancies and most importantly what type of information that data holds. The results of the EDA are presented in the Appendix

Having done an initial analysis of the data the next step is to create entities for the identified classes, object and data properties. As mentioned earlier the key information of the given dataset are menu items (pizzas) sold at restaurants across the US. Therefore, the key classes that will be created are ‘Restaurant’ and ‘Pizza’.

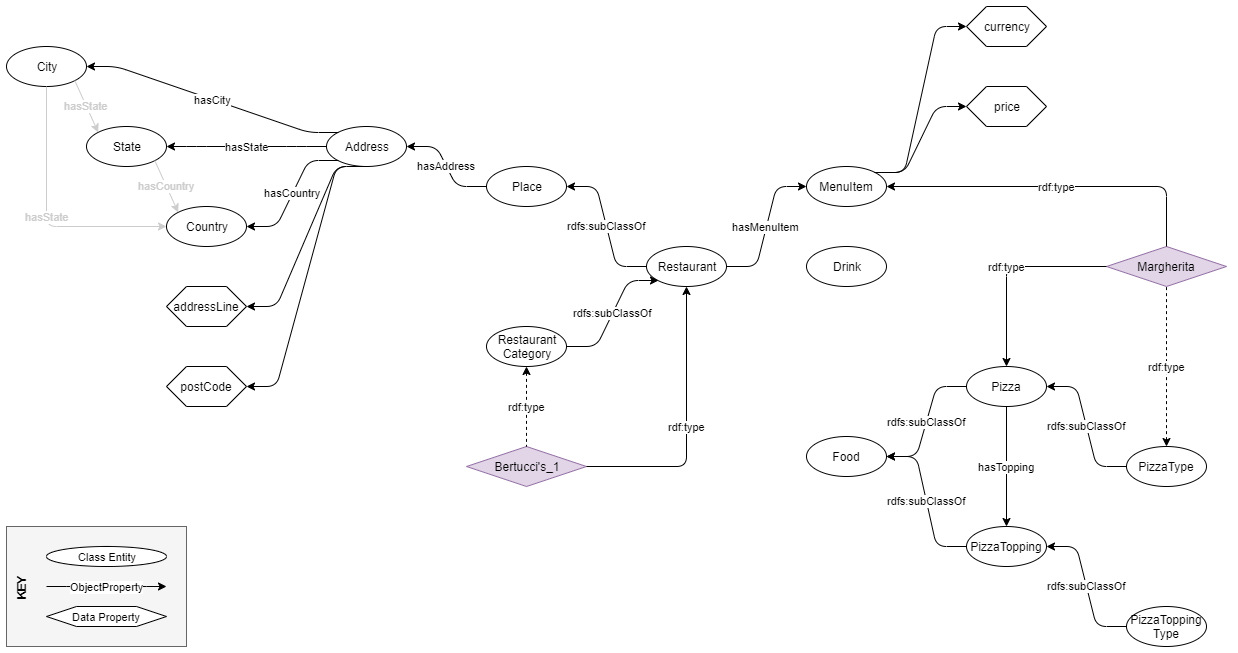


Figure 2. Ontology model for the tabular data

The diagram in Figure 2 presents an individual of the *Restaurant* class “Bertucci’s” and an individual of the *Pizza* class “Margherita” as well as all other entities consider for the ontology.

## Class Entities

### Place

The Place class and subclasses are used to model the details in the name and category columns. Although we know that the places referenced in the spreadsheet are all restaurants (i.e. part of the *Restaurant* class) an overarching parent class of *Place* has been created in case in the future we want to add other types of places like, museums, galleries etc. that share common object properties (e.g. hadAddress). The Restaurants are split further into multiple subclasses based on the information found in the categories column. Some of the categories are linked to an ethnic menu therefore they’ve been grouped under the *EthnicRestaurant* subclass like (*MexicanRestaurant*, *ItalianRestaurant*, etc.). Other categories are based on food type (e.g. *SeafoodRestaurant*, *PizzaRestaurant*, etc.) therefore they’ve are direct subclasses of the *Restaurant* class. The subclasses are not disjoined as based on some category strings a restaurant can belong to multiple categories represented by subtypes.

### Address / City / State / Country

Based on the data it was decided to make the *Address* an entity of each own rather than separate object properties of the *Place* class which would have been a simpler approach. However, having the *Address* being its own class, makes the model more flexible, in case an address house multiple restaurants (or more generally places). There are a few cases in the data we have that the same address houses two places:

e.g. ‘400 S Orlando Ave, 32751, FL’ is the address of ‘Nypd Pizza’ as well as ‘Francesco's Ristorante & Pizzeria’.

For city, state and country we only created classes in the zdetor ontology so that we can keep the original values from the spreadsheet as a name data property. Otherwise we wouldn’t need to create them in the ontology and we’d use the DBpedia URIs straight away. This is also the reason we’ve not created object properties to link cities to states and countries. In any case EDA has shown that the data is incorrect placing the same city in multiple different states.

### Food

We’ve created a *Food* parent class, disjoined with a *Drink* class in order to group together anything that can be classified as food. For the specific dataset we assume that all the menu items are pizzas. We’ve broken down the *Food* class in the *Pizza* and *PizzaTopping* subclasses.

*Pizza* is further broken down to multiple *PizzaType* subclasses which are retrieved in a similar way like the *RestaurantTypes* from 2.2.1.1. Pizzas with a specific name are all grouped together under the parent class *NamedPizza* which is a subclass of *Pizza*. Like the restaurant categories, given the quality of the data, the subclasses of *Pizza* has not been set up as disjoined classes

*PizzaToppings* and the subclasses follow the same pattern as with the *Pizzas.*

### MenuItem

The *MenuItem* has been set up as its own class so that we can have additional data properties like price, and currency. We decided to make this a separate class so that we can support in the future menu items that are not pizzas.

## Object Properties

### hasAddress / isAddressOf

The *hasAddress* is an object property that in the zdetor model links a *Place* instance to an *Address* instance. Given the *Restaurant* is a subclass on *Place* it will inherit that property. Based the definition of the restaurant, we make the *hasAddress* a functional property since a restaurant can only be located at one address. We’ve assumed that same restaurant names located at different addresses represent different individuals of the *Restaurant* class.

The *hasAddress* is the inverse of the *isAddressOf* property which based on the logic above is tagged as inverse functional.

Finally, we’ve only set the range and domain of the *hasAddress* and *isAddressOf* respectively to be of class *Address* but set the domain of the *hasAddress* locally in case we want to reuse the property for e.g. supplier addresses. The local setting is on the *Place* parent class with the axiom of ***hasAddress* some *Address***.

### hasMenuItem / isMenuItemOf

The *hasMenuItem* is an object property that in the zdetor model links a *Restaurant* individual to an *MenuItem* instance. Unlike the has address this is not a functional property since a restaurant can have multiple menu items. It’s inverse property, *isMenuItemOf* however is a functional object property since each menu item can only be served at one restaurant.

Moreover, for the *hasMenuItem* object property, we’ve set the domain and range globally since all subjects of this property will be *Restaurants* and all object will be *MenuItems*.

### hasCity / hasState / hasCountry

These object properties are used to add the details to the *Address* class. We’ve used them as existential axioms in the class instead of the global scope. One could argue for instance that *State* also is the domain of the *hasCity* therefore a global scope setting would make the *State* and *Address* equivalent.

Ideally instead of the existential we could’ve used the max cardinality since an address can only have up to one *City, State, Country* but we are making this existential so that the model is more flexible and can accommodate problems with the data

### hasIngredient / hasTopping / isIngredientOf / isToppingOf

These object properties have been set up in a hierarchy so that Ingredient is parent of topping. We do not have enough details to add more object properties (e.g. base as in the pizza owl).

The *hasIngredient* and *hasTopping* are the inverse properties of *isIngredientOf* and *isToppingOf* respectively. We’ve used the *hasTopping* as an existential axiom to link *Pizzas* with *PizzaToppings* and we’ve not made the property functional. More on that when we align the zdetor ontology with the pizza ontology.

### hasCurrency

The hasCurrency is used to link the *MenuItem* to a *Currency* individual. We’ve se this to max 1 because there are many menu items in the data that do not have a currency (or a price).

## Data Properties

For the data properties:

* *name* is reused by multiple classes,
* *postCode* and *addressLine* are specific to the *Address* entity
* *price, currency* and *description* are properties of the *MenuItem.* *Description* is there to old the actual value of the menu description even though we will also use the same field to identify some of the pizza toppings

# Tabular Data to Knowledge Graph [2.3]

Having established an initial version of the ontology the next step is to load to the data programmatically to a knowledge graph that is aligned with the classes, object and data properties we have established already.

The remaining of this section will describe the details of this process. Programmatically the code for this section is written in Python [cw3\_tabular\_data\_to\_KG.py]

## name

The name column is used to:

* create the URI for the *Restaurant* class but also
* represented as a data property (*name*).

As discussed in section 2.1 different restaurants share the same name therefore we cannot use it as is to create the URI. Instead, based on our assumption that the name-address is the composite key that uniquely identifies the restaurant, we create a new derivating column restaurant\_name that adds a suffix \_\_\_x to any restaurants whose names are shared. E.g.

|  |  |  |
| --- | --- | --- |
| **name** | **address** | **Restaurant\_name** |
| Bertucci's | 2929 Berlin Tpke | Bertucci's\_\_1 |
| Bertucci's | 194 Buckland Hills Dr | Bertucci's\_\_2 |
| Bertucci's | 2847 State Route 35 | Bertucci's\_\_3 |

Table 2. Examples or restaurant names reused for different restaurants

The incremental number in the suffix ensures that restaurant names are not unique. For restaurants that have unique names, the restaurant\_name is populated with the same value as the name column.

Programmatically we use the name column for the name data property and the restaurant\_name column for the URI creation. We perform some basic text processing of lower casing the string value and replacing some special characters before using the string value to create the URI. This processing is done for every string used to create the URI.

To avoid creating the same URI over and over again we’ve created the classStringToURI as a dictionary of the format: {'class':{'string': uri}}

The first key will be used to denote the class we are creating a URI for and the second would be a dictionary of key value pairs. That is because we may have the same value for a restaurant and a pizza, for instance, so we do not want to mix the two.

### address

Similar to the name, we use the address column to create both the URI for the *Address* class as well as the *addressLine* data property. To generate a unique identified for the address we concatenate the address and the state into a new ‘address\_id’ column. That is because there is the case of the following two addresses that share the same address line but seem to be different addresses altogether.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **address** | **city** | **country** | **postcode** | **state** |
| 222 E Main St | Mount Kisco | US | 10549 | NY |
| 222 E Main St | Collegeville | US | 19426 | Rahns |

Table 2. Examples or address lines reused for different addresses

## city

The city column is used to create the individuals for the zdetor:City class. Instead of generating new URIs for these value, we lookup the value of the columns in DBPedia and retrieve the most similar result. This process is not ideal the way it has been implemented and can be improved as part of future work

## country

This column is used to create the individual for the zdetor:Country class. There is only one individual needed (i.e. US) and the simple lexical matching lookup seems to have worked fine in this case and generated the following individual

dpo:United\_States a zdetor:Country .

## postcode

Post code is used to create the literal value for the *postCode* data property. For the few cases that the post code column is not populated we do not create an empty node with a blank value. We just skip the triple

## state

The state column is used to create the individuals for the *State* class in zdetor and the URIs are once again retrieved from dbpedia rather than generated. There’s some preprocessing taking place for the values from this column:

First, we load the state URIs from dbpedia in the classStringToURI[‘state’] dictionary with the key for each URI value being the 2 character iso code.

Then, given that the state column isn’t always the char-2 state code we derive a new column i.e. ‘state\_code’ that is populated as per the following pseudocode:

IF len(state) == 2  
THEN state\_code = state  
ELSE IF postcode is mapped to a state  
 THEN state\_code = state from postcode  
ELSE IF city is mapped to a state  
 THEN state\_code = state from city  
ELSE IF state\_code is NULL

Please note that in order for the above algorithm to work we’ve mapped us postcodes and us cities to states based on the dbpedia data.

Finally we use the char-2 state code from the ‘state\_code’ to find the URI from the classStringToURI[‘state\_code’] and use that to create the instance of the zdetor:State.

## categories

For the categories we performed a basic NLP for keyword extraction in order to identify the most frequent terms in the ‘categories’ column. The insights from the NLP results informed the creation of some subclasses of the *Restaurant* class. The list of subclasses is not exhaustive, and it was created after manual inspection the extracted keywords.

For the NLP we allowed up to 3 word per expression and also allowed for the same word to appear in multiple expressions. We also allowed for the restaurant subclasses in the ontology to be NOT be disjoined since a lot of restaurants in the csv have multiple categories. Finally we loaded the category subclasses from the ontology and used the same NLP in the category to match terms in the categories column with the class names. Whenever there was a match, we made the *Restaurant* URI a *RestaurantType* as well.

## menu item

We use this column to create the URI for the *Pizza* as well as the *name* object property. Before we generate the triple, however, we need to add one more column to the dataframe by concatenating the menu item and the restaurant name in order to create a unique field for URI. That is necessary because there are pizzas with the same name (i.e. menu item) that are served at different restaurants (i.e. if a pizza is served at 2 different restaurants has the same name we need to create to instances instead of one)

We also follow a similar logic as with the *RestaurantTypes* for the *Pizza* subclasses. For instance, if a menu item contains the term mozzarella, we use the URI to create the following 3 type triples.

zdetor:ak\_diamonds\_grilled\_margherita\_pizza

a zdetor:MargheritaPizza,

zdetor:MenuItem,

zdetor:Pizza ;

zdetor:name "Grilled Margherita Pizza"^^xsd:string .

## item value

We use this value, only when populated, to create the literal for the *price* object property.

## currency

We use this value, only when populated, to create the literal for the *currency* object property. We also reuse it to like the *Currency* individual to the *MenuItem* via the *hasCurrency* property.

## item description

Finally, we use this column, when populated to create the literal value for the *description* data property of the *MenuItem*.

We follow the same approach for text processing as with the categories and pizza types only in this case we use the identified triples to not only create subclasses for *ToppingTypes* but also individuals for those toppings. We create one individual for each *ToppingType* as follows:

zdetor:chicken a zdetor:ChickenTopping .

When processing the values of the item description in case we find a match to a topping we create a triple to link the topping individual to the *Pizza* via the *hasTopping* object property.

# Reasoning and SPARQL [2.4]

While performing the reasoning we found out that certain menu items had a multiple prices so we updated the price data property to not be functional. Otherwise the reasoning was doubling up prices 12 and 10 as well as 8 and 9. Programmatically the code for this section is written in Python [cw4\_reasoning\_and\_sparql.py]

## SPARQL.2

|  |
| --- |
| **SELECT** ?pizzaName ?description ?name ?addressLine  **WHERE**{  ?pizza a zdetor:Pizza;  zdetor:name ?pizzaName;  zdetor:isMenuItemOf ?restaurant;  zdetor:description ?description.  ?restaurant zdetor:name ?name;  zdetor:hasAddress [ zdetor:addressLine ?addressLine] .  **FILTER** (regex(LCASE(?pizzaName), \"white\") || regex(LCASE(?pizzaName), \"bianca\") || regex(LCASE(?pizzaName), \"bianco\") ).  # **FILTER NOT EXISTS** {  # **FILTER** regex(?description, \"tomato\").  # }  } |

Although we have a *WhitePizza* subclass we decided to not use that and instead try to lookup the terms white, bianca or bianco in the pizza name to get a few more individuals. For an additional refinement we added the not exists filter on the pizza description so that we can exclude any white pizzas that has tomato as a topping. The problem is that in a few instances the description has a negation (i.e. no tomato sauce) and negation is not handled by the script

## SPARQL.3

|  |
| --- |
| **SELECT** (sum(?price)/count(?pizza) as ?averagePrice) ?currency  **WHERE**{  ?pizza a zdetor:MargheritaPizza;  zdetor:price ?price;  zdetor:currency ?currency;  zdetor:isMenuItemOf [ zdetor:hasAddress [ zdetor:addressLine ?addressLine] ] .  }  **GROUP BY** ?currency |

For the average price I grouped by currency as well. In this case where all prices are in USD that grouping is redundant. However, if there were prices in other currencies as well then this grouping would ensure that only same currency prices would be summed up. In such a case we would get multiple results (i.e. one per currency).

Instead of filtering we just made the condition explicit that the pizzas should on type Margherita

## SPARQL.4

|  |
| --- |
| **SELECT** ?cityName ?state (count(?restaurant) as ?num\_of\_restaurants)  **WHERE**{  ?restaurant a zdetor:Restaurant;  zdetor:hasAddress ?address .  ?address zdetor:hasCity ?city . ?city zdetor:name ?cityName .    **OPTIONAL** {?address zdetor:hasState [ zdetor:name ?state] .}  }  **GROUP BY** ?cityName  **ORDER BY** ?state ?cityName |

Only interesting part of this script that needs mentioning is that we made the state optional since in some cases it is not populated with a valid dbo URI

## SPARQL.5

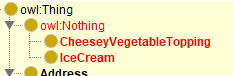
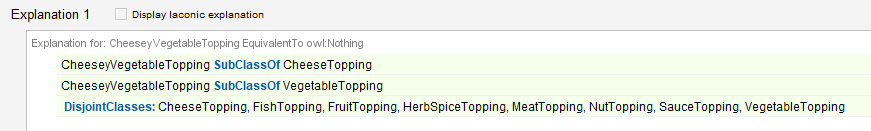
|  |
| --- |
| **SELECT** ?restaurant ?restaurantName ?postCode  **WHERE**{  ?restaurant a zdetor:Restaurant;  zdetor:hasAddress ?address ;  zdetor:name ?restaurantName .  ?address zdetor:hasCity ?city .  **OPTIONAL** {?address zdetor:postCode ?postCode .}  **FILTER**( !bound( ?postCode ) )  } |

As seen above the postcode is linked as optional to allow for NULL values and filter the rows were the postCode is not bounded i.e. is blank

# Ontology Alignment [2.5]

For ontology alignment we used a simple lexical matching of the names of the entities in the two ontologies (zdetor.ttl and pizza.ttl). Programmatically the code for this section is written in Python [cw5\_ontology\_alignment.py]. The code also allows for lexical mapping based on labels however the zdetor ontology doesn’t have extensive labels.

When running the reasoning without the data, and then loading the outcome [zdetor\_pizza\_alignment\_unreasoned.ttl] in protégé we only found some unsatisfiable classes in the pizza ontology.

As per the explanation the problem with the CheeseyVegetableTopping is that it is a subclass of both CheeseTopping and VegetableTopping and those 2 classes are disjoint.

When we tried to reason the KG with the data the code was running for a long time. Therefore we tested for a smaller set and realised that all pizzas having the same toppings were made equivalent. So, for instance when we tested with 50 pizzas it created 50^2 tripples. That is because in the pizza ontology the *isToppingOf* is a functional property, therefore pizzas with a same topping should really be the same pizza. This is contradicting with our assumption in the zdetor ontology where we only created one instance for each topping and reused for all pizzas with that ingredient. So we update the pizza ttl to remove the following highlighted in red and rerun the reasoner programmatically.

|  |
| --- |
| ### http://www.co-ode.org/ontologies/pizza/pizza.owl#hasTopping  pizza:hasTopping rdf:type owl:ObjectProperty ;  rdfs:subPropertyOf pizza:hasIngredient ;  owl:inverseOf pizza:isToppingOf ;  rdf:type owl:InverseFunctionalProperty ;  rdfs:domain pizza:Pizza ;  rdfs:range pizza:PizzaTopping ;  rdfs:comment "Note that hasTopping is inverse functional because isToppingOf is functional"@en . |
| ### http://www.co-ode.org/ontologies/pizza/pizza.owl#isToppingOf  pizza:isToppingOf rdf:type owl:ObjectProperty ;  rdfs:subPropertyOf pizza:isIngredientOf ;  rdf:type owl:FunctionalProperty ;  rdfs:comment "Any given instance of topping should only be added to a single pizza (no cheap half-measures on our pizzas)"@en . |

After running the reasoning with the data and got the asserted triples we run a sparql query to find meaty pizzas. The script returned results even though we had not created the class *MeatyPizza.* That is because:

* *pizza:MeatyPizza* is a defined class [pizza:*Pizza* and (pizza:*hasTopping* some pizza:*MeatTopping*)
* Also the *pizza:MeatTopping* has the following subclasses *pizza:ChickenTopping*, *pizza:HamTopping*, *pizza:HotSpicedBeefTopping*, *pizza:PeperoniSausageTopping*.
* From the alignment *Pizza*, *hasTopping*, *ChickenTopping* and *HamTopping* have all been matched like for like between the pizza and the zdetor ontology

As a result any pizza instance that has a *zdetor:HamTopping* and/or a *zdetor:ChickenTopping* has an inferred *pizza:MeatTopping* and as a result belongs to the *pizza:MeatyPizza*.

# Ontology Embeddings [2.6]

# Appendix

## Exploratory Data Analysis (EDA)

Following is a high-level signature of the dataset

|  | **Description** |
| --- | --- |
| ***Number of Rows:*** | 3,510 |
| ***Grain:*** | The grain of the data in the csv file is at the menu item level. This means that each row represents a menu item (a pizza in the majority of the cases). There are also columns pertaining to the restaurant that serves each of those menu items. This information has been de-normalised and appears with the same set of values across all items served at each restaurant. |
| ***Columns:*** |  |
| name | *Number of unique restaurant names*: 933  Exploratory data analysis has shown that the name doesn’t uniquely identify the restaurant since there are a few sites with multiple addresses (e.g. post codes) and each address has it’s own set of menu items.  For instance, the site name Bertucci's appears with 4 different addresses and although three of them appear to be of the same franchise (i.e. having similar menu items with the same description and price), one of the three is missing the Margherita Pizza. Moreover the 4th one, the one in Illinois seems to be a completely different restaurant since it only has one menu item that is different from the menu items of the other 3 Bertucci’s restaurants.  The above observations have highlighted the need to combine the name with at least one more column before we can create a new instance of a site otherwise the data would incorrectly assume that because the name is the same all 4 Bertucci's sites are actually the same site.  Also, it has rendered the assumption that all restaurants sharing the same name are part of the same franchise/chain incorrect. Therefore, there is not enough information to safely create a hierarchy of restaurants belonging to the same chain (e.g. Bertucci’s, PapaJohn’s, Applebee’s) |
| address | The address line of the address the restaurant is in. This field, same as the name, is always populated so is a good candidate to use for a URI describing the restaurant |
| city | The city the restaurant is in |
| country | The country the restaurant is in. This column is always populated with the value ‘US’ which means all restaurants in the csv file are located in the United States |
| postcode | The address post code  Nulls: 65 |
| state | The state the restaurant is in. This column appears to have data of all sorts, so we decided to investigate the data a bit further. To avoid noise from the menu items, we focused on the section of the data that described the address and found the following:  *Number of unique addresses*: 989  *Unique values for state*: 281  *State values not 2 characters long*: 29.97%  *State values exactly 2 characters long*: 70.03%  Given that all the addresses are in the US and our prior knowledge that there are 52 states in the US, the 281 unique values in the state column imply that the column is sometimes used to store other type of information.  Given that the majority of the addresses (i.e.70%) have the char-2 state code, we will try to clean the remaining 30% during the creation of the URIs |
| categories | This column has some textual description of the category the restaurant belongs to. |
| menu item | The name of the menu item. This field is always populated so we can used it for the URI creation. However, the values are repeated across multiple restaurants (similar to the restaurant name) so we need to be careful when creating the individuals |
| item value | Nulls: 562  Not all items have prices. However, we are making the decision not to filter out any of the items missing a price as they may be offered for free. Moreover, we do not want to miss any of the information from the spreadsheet |
| currency | Nulls: 559  Same as the null values for the item values which goes to show that whenever we have a price, we also have a currency. It this was not the case we may choose to infer the currency from the country of the restaurant, but we will not do that for this case |
| item description | Nulls: 1984  A short description of the menu item. In most cases this column has a lot of useful details on the pizza ingredients. |

Table 1. Dataset signature

## Entities in the zdetor ontology

The complete list of entities created for the ontology, which will hereafter me referred to as *zdetor* is specified in Table 2.

|  |  |  |
| --- | --- | --- |
| **Classes** | **Object properties** | **Data Properties** |
| Place  Restaurant  RestaurantType  EthnicRestaurant  RestaurantType | hasAddress  hasMenuItem | name |
| Address  City  State  Country  Currency | hasCity  hasState  hasCountry  isAddressOf | postCode  addressLine |
| Drink  Food  Pizza  PizzaType  NamedPizza  PizzaType  PizzaTopping  PizzaToppingType | hasIngredient  hasTopping  isIngredientOf  isToppingOf |  |
| MenuItem | isMenuItemOf  hasCurrency | price  currency  description |

Table 2. Entities in the zdetor ontology