Assignment 4: Knowledge Graph Quality

Mater Student Evangelia P. Panourgia, f3352402
Professor Dr. Panos Alexopoulos

Replication Code

The zip file contains a **README.md** file that provides an overview of its structure.

Task 1

For the scope of Task 1, we have decided to divide it into two parts. Part A will focus on analyzing ambiguity in single words, while Part B will address ambiguity in phrases (more than 2 words). This separation is due to the fact that we will handle phrases differently from individual words.

Part A: Handling One Word

In Part A, we focus on generating one-word labels across three primary categories: class names, attribute names, and relation names. Each label must adhere to specific part-of-speech requirements to support a robust analysis of ambiguity, particularly as we aim to mitigate the influence of labels with similar structures that could skew the results.

This analysis arises from observed inconsistencies in DBpedia naming practices, where class and attribute names do not consistently follow best practices. Specifically, DBpedia often employs parts of speech other than nouns for class and attribute labels, and it uses verbs and other parts of speech interchangeably for relation names. In our study, we aim to avoid these inconsistencies by adhering to best practices: using nouns exclusively for class and attribute names and verbs exclusively for relation names.

For example, consider the word "mine." In a dictionary, "mine" has meanings as both a noun and a verb. However, if best practices in naming conventions were followed, it would appear only as a noun in the context of class or attribute names. By following these guidelines, we ensure that our analysis remains focused on samples aligned with best practices, avoiding the potential ambiguity introduced by multi-part-of-speech terms.

Thus, this analysis aims to explore a sample based on the best practices for naming conventions, ultimately enhancing the quality of ambiguity analysis by standardizing the linguistic structure of our dataset.

Figure 1 illustrates the data pipeline followed for the selection and processing of class labels. Initially, we executed a query to retrieve all class labels used in DBpedia. Next, we filtered this list to retain only those labels categorized as nouns, based on their part of speech. Subsequently, we removed any labels containing camel case words (e.g., *DataScience*), as these represent compound terms rather than single words.

To determine the required sample size for statistical generalization, we applied Cochran's formula, allowing us to estimate the number of labels necessary to examine. Using this calculated sample size, we then removed these labels from the dataset to create a secondary sample. This smaller sample serves as input for few-shot learning examples in our language model (LLM) training process. Above each arrow in Figure 1, we display the number of labels present at each stage of the pipeline.

Following a similar approach, we created the attribute and relation label datasets, as illustrated in Figures 2 and 3. Each figure shows the number of labels at each stage, detailing the steps taken to refine the labels for use in downstream tasks.

Note: For relation labels, we included labels containing verb with prepositions, as these are not considered to be multiple-word terms in this context (e.g. play in).

In DBpedia, **properties** consist of **both attributes and relations**. To differentiate between them, we verify the 'rdfs:range' of each property. If the range has a datatype (e.g., 'xsd:nonNegativeInteger'), we classify it as an attribute; otherwise, it is considered a relation. Figure 1 presents the SPARQL query code used for this differentiation.

```
 # Query 2: Retrieve Attribute Names (properties with specified XML Schema datatype
    ranges)
query_attributes = """
SELECT DISTINCT ?label
WHERE {
  ?property a rdf:Property .
  ?property rdfs:label ?label .
  ?property rdfs:range ?range .
  FILTER (lang(?label) = 'en')
  FILTER (regex(?label, "^[A-Za-z]+$", "i"))
  FILTER (
      ?range IN (
          xsd:string, xsd:integer, xsd:nonNegativeInteger, xsd:float,
          xsd:double, xsd:date, xsd:dateTime, xsd:boolean, xsd:decimal,
          xsd:time, xsd:duration, xsd:gYear, xsd:gYearMonth, xsd:gMonth,
          xsd:gMonthDay, xsd:gDay
      )
  ) # Ensures the range is one of the specified XML Schema datatypes
}
"""

# Query 3: Retrieve Relation Names (properties with object ranges) with one-word or
   two-word labels
query_relations = """
SELECT DISTINCT ?label
WHERE {
  ?property a rdf:Property .
  ?property rdfs:label ?label .
  ?property rdfs:range ?range .
  FILTER (lang(?label) = 'en')
  FILTER (
      regex(?label, "^[A-Za-z]+$", "i") ||
      regex(?label, "^[A-Za-z]+ (in|on|to|for|by|with|at|from|about|as|into|like|
          through|after|over|between|under|among)$", "i")
  )
  FILTER (!(
      ?range IN (
          xsd:string, xsd:integer, xsd:nonNegativeInteger, xsd:float,
          xsd:double, xsd:date, xsd:dateTime, xsd:boolean, xsd:decimal,
          xsd:time, xsd:duration, xsd:gYear, xsd:gYearMonth, xsd:gMonth,
          xsd:gMonthDay, xsd:gDay
      )
  )) # Ensures range is not one of the specified XML Schema datatypes
}
"""
```

Listing 1: SPARQL Query for Differentiating Attributes and Relations in DBpedia Properties

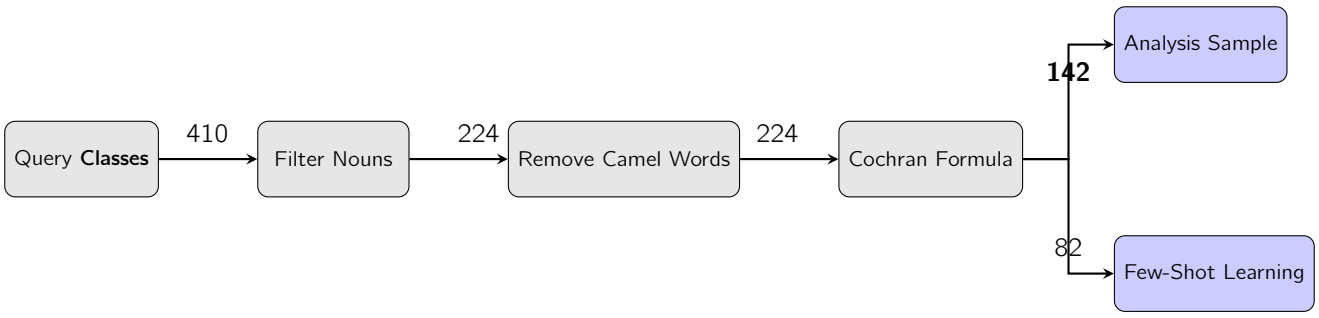


Figure 1: Diagram of Class Labels Processing

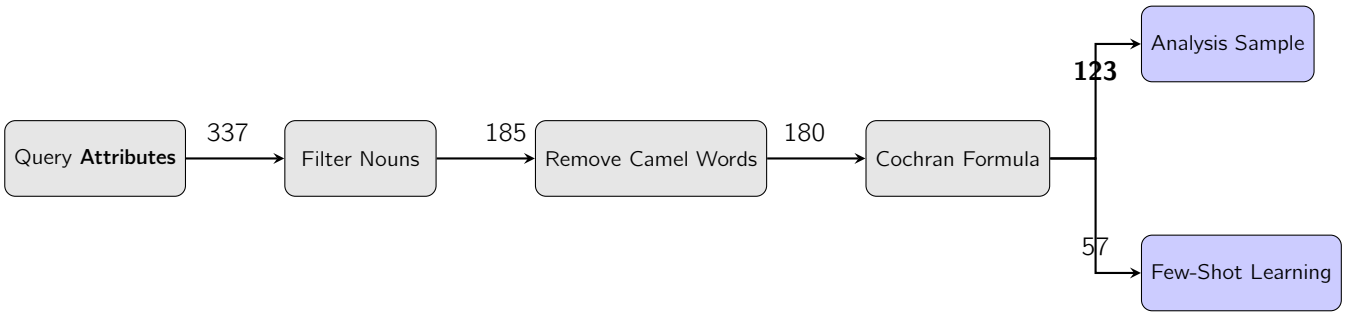


Figure 2: Diagram of Attributes Processing

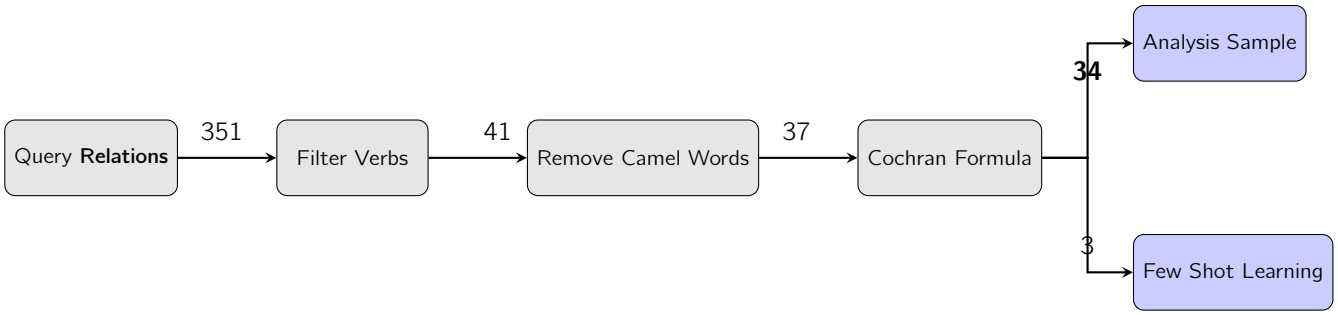


Figure 3: Diagram of Relations Processing

Note: The presence of low-quality labels in DBpedia is evident in the pipeline stages in Figures 1, 2, 3, as indicated by the numbers above the arrows between the "Filter Nouns/Verbs" and "Remove Camel Words" stages. Specifically, 186, 152, and 310 labels were identified as inappropriate types regarding the "best" part of speech for the categories of classes, attributes, and relations, respectively. These figures underscore the filtering and refinement necessary to obtain high-quality, relevant labels for each category.

Note: The following analysis assesses label quality solely for those labels that adhere to established best practices for naming conventions. Labels that did not meet these standards, as identified in earlier filtering stages, were excluded from this quality evaluation. It is essential to ensure that a single poorly named property does not disproportionately impact the quality metric of ambiguity. Therefore, our approach considers a balanced aggregation of names, mitigating the effect of outliers on the overall ambiguity assessment.

Dictinary Approach and LLM for Class Labels having one Word: Having prepared our sample data for class labels, we read it into a list structure and pass each element of the list to the function `analyze_words` (Figure 2). The `analyze_words` function is designed to analyze a list of words by retrieving definitions and parts of speech (POS) for each meaning, specifically from the WordNet lexical database.This function takes two parameters: `word_list`, which is a list of words to be analyzed, and `pos`, an optional parameter to filter results by part of speech. The function searches WordNet for synsets related to each word, where each synset represents a unique meaning and context. The definitions are then organized in a dictionary structure where each word maps to a list of meanings, with each meaning represented as a dictionary containing its part of speech and definition. If `pos` is specified, only definitions with that specified POS (e.g., 'n' for nouns) are included in the output.

```
def analyze_words(word_list, pos=None):
    """
    Analyzes each word in the list, providing definitions and parts of speech for
        each meaning.

    :param word_list: List of words to analyze
    :param pos: Part of speech to filter by (optional).
                Accepts 'n' (noun), 'v' (verb), 'a' (adjective), 'r' (adverb).
    :return: Dictionary where each word maps to a list of dictionaries containing POS
         and definition.
    """
    analysis = {}

    for word in word_list:
        # Get synsets for the word and specified POS
        synsets = wn.synsets(word, pos=pos)

        # Collect definitions and POS for each synset
        meanings = []
        for synset in synsets:
            meaning_info = {
                "part_of_speech": synset.pos(),
                "definition": synset.definition()
            }
            meanings.append(meaning_info)

        # Map word to its list of meanings (empty if no meanings found for specified
            POS)
        analysis[word] = meanings

    return analysis

class_analysis = analyze_words(sample_class, pos='n')  # call function by limitting
    with n = noun
```

Listing 2: The `analyze_words` function

After generating definitions with the `analyze_words` function, we proceed by calling it with the code in Figure 3.

```
class_analysis = analyze_words(sample_class, pos='n')  # Change pos as needed
list_class = {key: len(value) for key, value in class_analysis.items() if isinstance(
    value, list)}
df_class = pd.DataFrame(list(list_class.items()), columns=['class_label_noun_name', '
    number_of_noun_definitions'])

df_class['dictionary_label'] = df_class['number_of_noun_definitions'].apply(lambda x:
     1 if x > 1 else 0)
```

Listing 3: Creating a DataFrame of Class Labels and Noun Definitions

In this code, we use the function `analyze_words` on a sample list of class labels, setting `pos='n'` to focus specifically on noun definitions. The resulting dictionary, `class_analysis`, maps each word in the list to a collection of meanings. To count the number of noun definitions per word, we construct a dictionary comprehension `list_class` where each word is mapped to the length of its list of meanings (i.e., the number of noun definitions).

Finally, we convert this dictionary into a DataFrame, `df_class`, for structured data analysis. The DataFrame is constructed with two columns: `class_label_noun_name`, containing the words analyzed as class labels, and `number_of_noun_definitions`, containing the count of noun definitions associated with each word. This organized data format facilitates further exploration and analysis of the class labels and their meanings. Furthermore, we used the `apply` function with a lambda expression to evaluate each value in the `number_of_noun_definitions` column. For each value, it assigns a label of 1 if the value is greater than 1, indicating that the word has multiple noun definitions. Otherwise, it assigns a label of 0, signifying that the word has only one or no noun definitions. This new column, `dictionary_label`, effectively serves as a binary indicator, helping to differentiate between

words with single or multiple noun meanings. The resulting `df_class` DataFrame now contains an additional layer of information, providing insights into the semantic richness of each class label.

Regarding the use of the LLM (Large Language Model) approach for assessing ambiguity in class labels, we designed a prompt containing carefully selected examples to enable few-shot learning. The examples, pulled from the sample dataset shown in Figure 1, guide the model in identifying ambiguous terms effectively. This prompt contains both ambiguous and non-ambiguous examples, which illustrate terms with multiple noun meanings to indicate ambiguity. In the following box, it is demonstrated the design of this prompt.

After defining the prompt, we extended the existing `df_class` DataFrame by adding a new column named `llm_label`, which records the model's ambiguity assessment for each term. This new column assigns a value of 1 for terms with multiple noun definitions (indicating ambiguity) and 0 otherwise, facilitating a structured analysis of the class labels' semantic complexity.

Having implemented both approaches for investigating ambiguity in single-word class labels, we utilized two distinct methods: the dictionary-based approach (WordNet) and the LLM prompt-based approach (few-shot learning). Using the dictionary approach, we identified ambiguous terms based on multiple noun definitions provided by WordNet. Meanwhile, the LLM prompt approach involved a few-shot learning prompt to evaluate ambiguity, leveraging examples to guide the model's responses.

After conducting analyses with both methods, we calculated the percentage of ambiguous terms in our sample of single-word class labels. This percentage provides insight into the semantic variability within the class labels and highlights terms with potentially broad or multiple interpretations.

In Table 1, the first row displays the percentages of ambiguous terms within our sample. Observing both percentages, it is evident that the majority of single-word noun class label names exhibit ambiguity. This result implies that the ontology may suffer from poor quality concerning this metric, as high ambiguity can hinder clarity and precision in classification.

Note that regarding the performance of the LLM, we observe a difference of 10 percentage lower points compared to the dictionary approach. This indicates that the LLM is not entirely precise, as its results differ from the dictionary-based approach, which is more grounded in actual lexical data. **The LLM's responses may exhibit variability due to the inherent stochastic nature of large language models, leading to less reliable results when compared with the dictionary approach**. However, a 10-unit difference is not substantial, indicating that the LLM demonstrates relatively good performance.

	Dictionary Approach	**LLM-Prompt Approach**
Class Label	72%	62%
Attribute Label	78%	86%
Relation Label	32%	47%

Table 1: Percentages of Ambiguity Terms Across Classes, Properties, and Approaches - 1 Word

Dictionary Approach and LLM for Attribute Labels having one Word: Similarly to the approaches used in the dictionary and LLM prompt methods for class labels containing a single word, we applied these methods to our samples related to attributes. Since attribute names, like class names, are also nouns, we used the same few-shot learning prompt as with the class label analysis.

In the dictionary approach, we analyzed the noun definitions of each attribute term, assigning ambiguity labels based on the number of noun meanings. For the LLM prompt-based approach, we utilized the few-shot learning strategy as implemented with class labels, providing the model with examples that frame nouns as attributes to guide its interpretation. This consistent methodology allowed for parallel analysis, facilitating a direct comparison of ambiguity in both class labels and attribute names.

In Table 1, the second row presents the calculated percentages for attribute names, indicating the coverage percentage of ambiguity within our sample. This provides insight into the extent to which attribute names exhibit ambiguous meanings, as identified by both the dictionary and LLM approaches. Regarding the comparison between the dictionary approach and the LLM, we observe that the LLM consistently labels a higher percentage of terms as ambiguous, often incorrectly. This suggests that the LLM tends to overestimate ambiguity in attribute names (86%) compared to the more precise dictionary approach (78%). However, an 8-unit difference is not considered significant, suggesting that the LLM again demonstrates good performance. Nevertheless, the ontology still shows ambiguity concerning the attribute labels.

Prompt for Ambiguity Assessment: Class and Attribute Labels: One Word

```
prompt = (
    f"You are an expert in linguistic analysis. Your task is to determine "
    f"if the term '{term}' is ambiguous.\n\n"

    "A term is considered ambiguous if it has multiple meanings as a **noun**. "
    "Each meaning with the part of speech labeled as **'n'** is considered "
    "a noun definition.\n\n"

    "Please answer **'Yes'** if the term has more than one noun definition, "
    "meaning it is ambiguous, "
    "or **'No'** if it has one or no noun definitions, meaning it is not ambiguous.\n\n"

    "Here are examples to help you understand:\n\n"

    "- **Ubiquitous Examples (Answer: Yes)**:\n"

    "  - Term: \"church\"\n"
    "    - Definition 1: (n) \"a building used for public worship, especially Christian\" "
    "(e.g., a place of worship in many communities)\n"
    "    - Definition 2: (n) \"an organized body of religious believers\" "
    "(e.g., the institution or organization as in 'the Church')\n\n"


    "  - Term: \"sales\"\n"
    "    - Definition 1: (n) \"the exchange of a commodity for money\" "
    "(e.g., retail transactions, sales events)\n"
    "    - Definition 2: (n) \"a business or occupation of selling goods or services\" "
    "(e.g., roles like a sales manager or salesperson)\n\n"

    "- **Not Ubiquitous Examples (Answer: No)**:\n"
    "  - Term: \"cyclist\"\n"
    "    - Definition 1: (n) \"a person who rides a bicycle\" "
    "(e.g., involved in sports or recreational cycling)\n\n"

    "  - Term: \"airport\"\n"
    "    - Definition 1: (n) \"a complex of runways and buildings for the takeoff,

    landing, and maintenance of aircraft\" "
    "(e.g., places like JFK Airport, used mainly in travel contexts)\n\n"
)
```

Note: In the actual prompt, additional examples are included. Here, only two examples per category are shown to conserve space, though the full prompt contains five examples in each section.

Dictionary Approach and LLM for Relation Labels having one Word:
Similarly to the approaches used in the dictionary and LLM prompt methods for class and attribute labels containing a single word, we applied these methods to the relation labels. The primary difference here is the part of speech; instead of focusing on nouns, we analyzed verbs and used related examples of verb terms. The prompt for the LLM (few-shot learning) is presented in the box below.

Furthermore, in Table 1, we observe that the LLM again labels more terms incorrectly as ambiguous compared to the dictionary approach. Specifically, the dictionary approach reveals that 32% of the terms in relation labels exhibit ambiguity, reflecting the actual percentage. This trend of the LLM yielding a higher percentage of ambiguity (47%) continues across categories. However, again the LLM has good performance as it is not significantly different from dictionary-based percentage metrics. It is worth noting that the ontology `DBpedia` seems to have less ambiguity in the relations compared to attributes and classes.

In conclusion, to provide an overall comparison of class labels, attribute labels, and relation labels for single-word terms, we created Figure 4. This figure illustrates that the LLM tends to correctly classify most terms as either ambiguous or not, and we can confirm this since we know the correct answer through the dictionary approach, which is not based on stochastic or auto-generated answers that may be prone to error. Naturally, the dictionary approach offers a more accurate view of ambiguity percentages. Based on this, rather than on the LLM, we observe that class and attribute labels exhibit high ambiguity, whereas relation labels show relatively less.

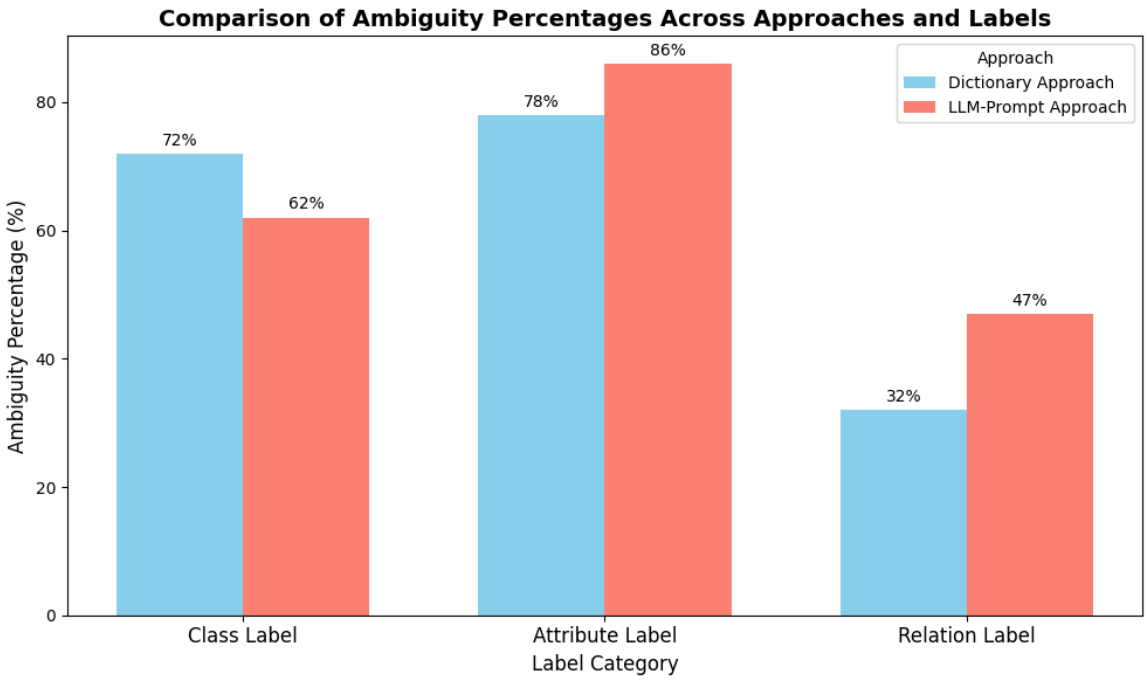
It is important to note that if this analysis were not limited to part-of-speech distinctions, the percentages of ambiguity would likely be even higher. So, it might be a special case of analysis, but we know exactly which part of speech we are examining. Thus, our analysis provides generalized insights, supported by the Cochran formula, focusing on ambiguity percentages specifically for class labels as nouns, attribute labels as nouns, and relation labels as verbs. As a general "rule" for names, if we encounter a term like mine, we avoid translating it with a definite, non-noun meaning. Instead, we interpret it according to its dual noun definitions, following established best practices for naming conventions. So, we tried to respect this rule in our analysis. Thanks to this analysis, we identified an additional obstacle in the ontology: a failure to adhere to best naming practices regarding parts of speech, which represents an additional issue of quality. In Figures 1, 2, and 3, observe the number of arrows between the rectangles labeled Filter Noun / Verb and Remove Camel Word. This indicates that a significant number of terms do not follow the best practice for part-of-speech consistency.

So the aforementioned analysis based on **Lexical Ambiguity: When a word has multiple meanings.**

Note: When it comes to resolving ambiguity in single-word detection, a dictionary-based approach is generally preferable to using a language model (LLM). This is because **a dictionary approach relies on a predefined database of terms, offering reliable, accurate results. Additionally, our online dictionary includes information on multiple definitions and parts of speech, which minimizes the risk of incorrectly categorizing a term as ambiguous when we actually intend to limit the scope for example to noun definitions. This structured data provides clarity and precision. So, we would say that dictionary approach is more efficient than LLM approach as we can trust it.**

While in our specific case, the LLM demonstrates good performance (as it seems to "understand" the task and limitation of part of speech), we maintain that a dictionary approach is typically more efficient and accurate. This is due to its reliance on established definitions rather than the probabilistic, auto-generated responses characteristic of LLMs.

In the Figures 5, 6, 7, we present the **most ambiguous terms among class, attribute, and relation labels**. These terms, distinguished by their multiple definitions, illustrate the complexity and variability in meaning associated with each corresponding part of speech.



Note: The LLM-Prompt Approach consistently shows higher ambiguity percentages than the Dictionary Approach.

Figure 4: Distribution of Incorrect Class Assignments by Percentage

Ambiguity Assessment Prompt : Relation Labels : One Word

You are an expert in linguistic analysis. Your task is to determine if the term `'{term}'` is ambiguous. A term is considered ambiguous if it has multiple meanings as a **verb**. Each meaning with the part of speech labeled as **'v'** is considered a verb definition.

Please answer **'Yes'** if the term has more than one verb definition, meaning it is ambiguous, or **'No'** if it has one or no verb definitions, meaning it is not ambiguous.

Here are examples to help you understand:

Ambiguous Examples (Answer: Yes):
- Term: **promoted**
- Definition 1: (v) *to advance in position or rank*
- Definition 2: (v) *to advertise or publicize something*
- Term: **editing**
- Definition 1: (v) *to modify text or media for clarity and precision*
- Definition 2: (v) *to prepare text or media for publication by making revisions*

Based on this information, respond with **'Yes'** if the term `'{term}'` is ambiguous (multiple verb meanings) or **'No'** if it is not.

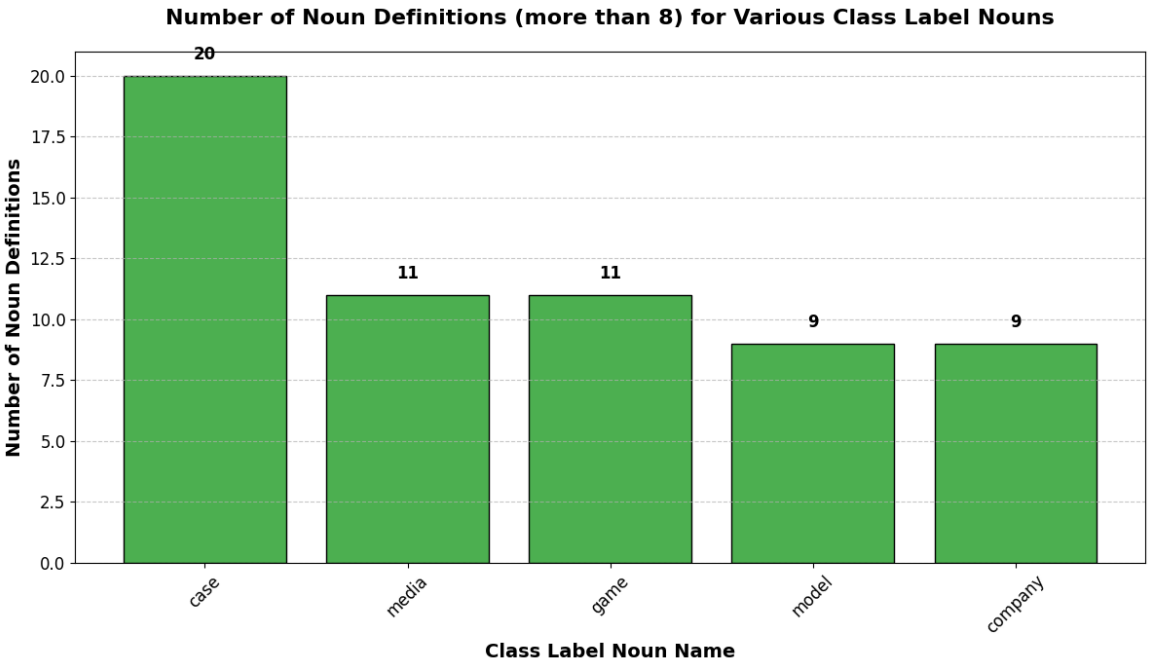


Figure 5: Distribution of Incorrect **Class** Assignments by Percentage

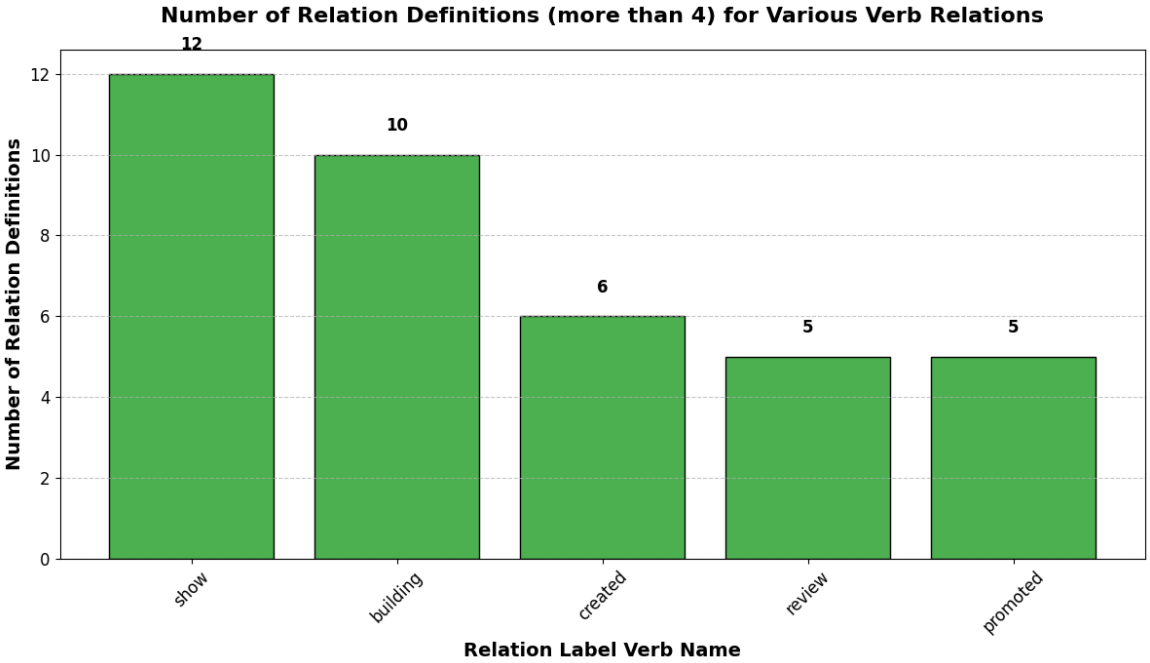


Figure 6: Distribution of Incorrect **Relation** Assignments by Percentage

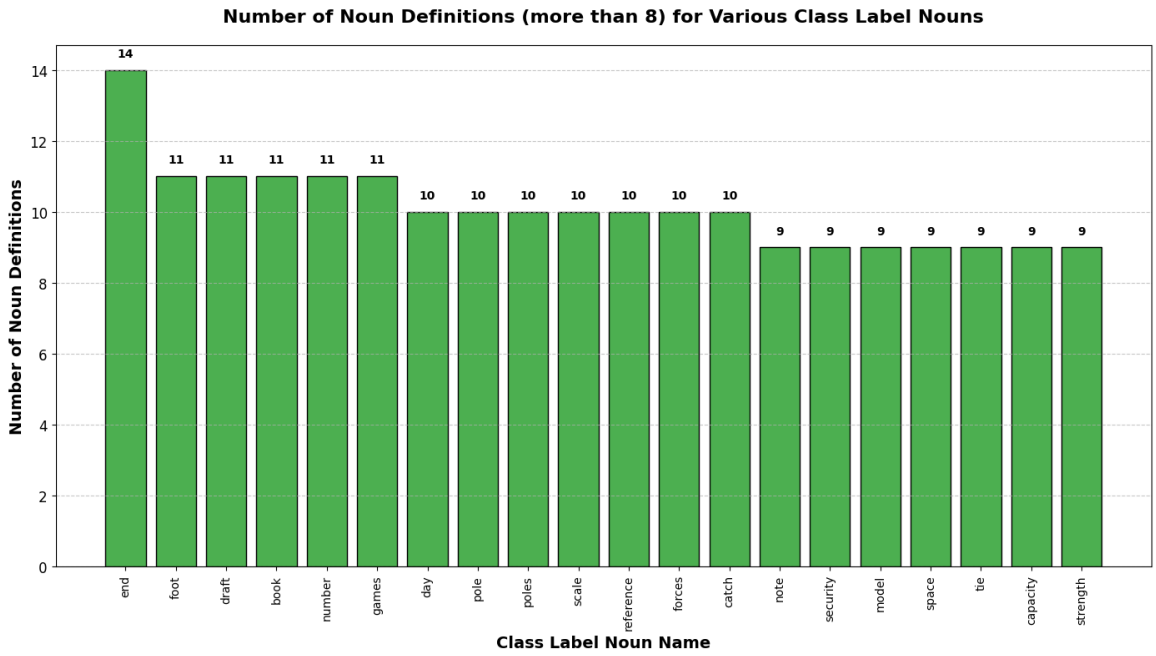


Figure 7: Distribution of Incorrect **Attribute** Assignments by Percentage

Part B: Handling Phrases

In this section, we analyze class labels, attribute labels, and relation labels in three dimensions, focusing on phrases of at least three words compared to single-word terms (we defined three words instead of two, as we anticipated cases where data might appear in the format "play in," where the second word is a preposition. Including a third word helps smooth the results by capturing such variations. This approach was also integrated to better relate to single-word queries, see relation labels for one word in the previous section). This shift in focus is due to the fact that the dictionary approach is less applicable to multi-word phrases. Consequently, heuristic methods, as well as different few-shot learning examples and tailored prompts, are necessary for this analysis.

By exploring multi-word phrases, we aim to capture nuances in ambiguity that single-word approaches may overlook, thereby providing a more comprehensive understanding of ambiguity across DBpedia's labels.

Firstly, to create our dataset for each dimension, we retrieve labels from DBpedia that contain at least three words. The code utilizes the SPARQLWrapper library to interact with the DBpedia SPARQL endpoint, allowing us to execute SPARQL queries and extract labels in English.

The code defines a function, `fetch_labels`, which takes a SPARQL query as input, executes it, and returns the labels from the results. We formulate three separate SPARQL queries to target different types of labels:

- **Class labels**: These are retrieved by querying resources that are instances of `owl:Class`. The filter criteria ensure that only English labels with at least three words are selected.
- **Relation labels**: Retrieved by querying resources that are instances of `rdf:Property`. Similarly, only English labels with three or more words are included. Additionally, to distinguish labels related to attributes from relation labels, we applied a filter based on range type. If the range was a datatype (e.g., `xsd:string`, `xsd:integer`), it was classified as an attribute label; otherwise, it was classified as a relation label.
- **Attribute labels**: Also retrieved from resources classified as `rdf:Property`, applying the same filters for English and multi-word labels. Additionally, to distinguish labels related to attributes from relation labels, we applied a filter based on range type. If the range was a datatype (e.g., `xsd:string`, `xsd:integer`), it was classified as an attribute label; otherwise, it was classified as a relation label.

As is evident from Figures 8, 9, and 10, we performed basic preprocessing by applying **Cochran's formula** to determine an appropriate sample size for **statistical generalization** (that is 72, 246 and 117 for class , attributes and relations phrases respectively). This allowed us to analyze only a specific subset that is statistically representative of the entire dataset. The remaining elements were saved to respective lists for potential use as examples in

few-shot learning definitions, should additional examples be required.

In contrast, for the single-word analysis, we will not consider part of speech, as it does not make sense in the context of phrase analysis.

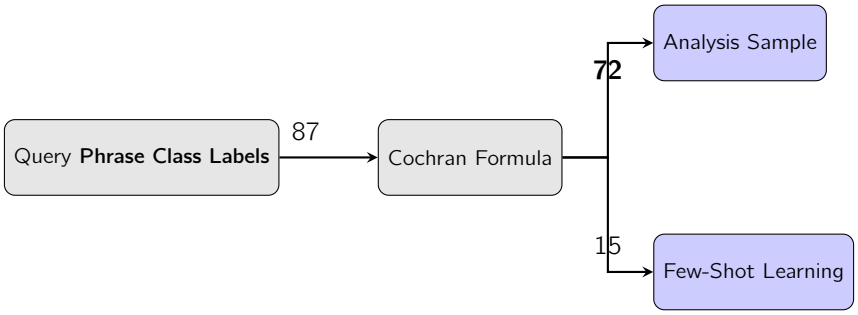


Figure 8: Diagram of Class **Phrase** Labels Processing

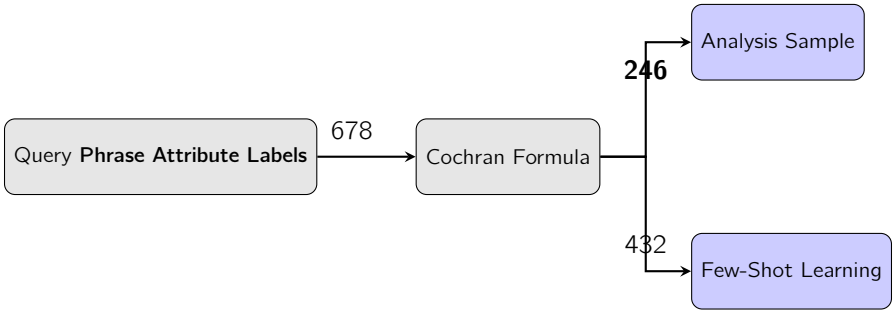


Figure 9: Diagram of Attribute **Phrase** Labels Processing

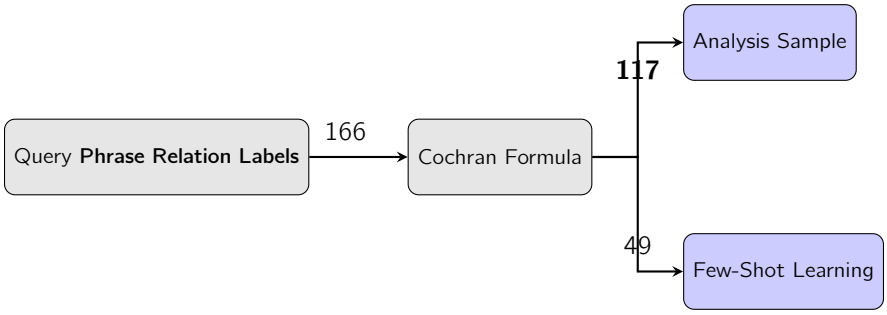


Figure 10: Diagram of Relation **Phrase** Labels Processing

The heuristic and method-based approaches for analyzing phrases are similar to class attributes and relational structures, so we will provide a unified explanation that applies to all.

Regarding the heuristic approach, the code in Figure 4 begins by importing necessary libraries and models to perform intrinsic ambiguity detection on phrases. The `pandas` library manages data structures such as DataFrames, allowing for efficient data manipulation. The `transformers` library loads a pre-trained T5 model and tokenizer, specifically fine-tuned for paraphrasing tasks. The `torch` library provides support for tensor operations required by the model, while `cosine_similarity` from `sklearn.metrics.pairwise` calculates the similarity between embeddings, helping to quantify phrase ambiguity.

The T5 model and tokenizer are initialized using the `T5Tokenizer.from_pretrained` and `T5ForConditionalGeneration.from_pretrained` methods, allowing the model to generate paraphrases, or alternate expressions, for a given phrase.

The `get_embedding` function generates a vector representation, or *embedding*, for each phrase. This function tokenizes the text and passes it through the T5 encoder, where the embedding is computed by averaging the hidden states from the encoder's last layer, creating a fixed-length vector. This embedding enables comparisons between the original phrase and its paraphrased variations.

The `generate_paraphrases` function produces multiple paraphrases by prompting the model with a prefixed phrase, "paraphrase: ". Beam search, used here, generates multiple paraphrase options, controlled by the `num_paraphrases` parameter. The outputs are decoded into a list of paraphrased phrases.

The `detect_intrinsic_ambiguity` function determines the ambiguity of a phrase by examining the similarity of its paraphrases. After generating paraphrases and obtaining embeddings, the function calculates similarity scores between the original phrase and each paraphrase using `cosine_similarity`. The range of these similarity scores, measured as the difference between the highest and lowest scores, indicates the ambiguity level. If this range exceeds a threshold (0.2 in this case), the function classifies the phrase as ambiguous (returning 1). Otherwise, it is deemed unambiguous (returning 0).

The `add_ambiguity_column` function applies the `detect_intrinsic_ambiguity` function across each entry in a DataFrame column, enabling batch processing. A new column is added to the DataFrame where each row shows the ambiguity status of its respective phrase.

To illustrate, let's consider a DataFrame, `df_sample_class_phrase`, containing two phrases:

Phrase
"The bank is closed"
"He pitched the idea"

Table 2: Sample DataFrame of Phrases

- For the phrase "The bank is closed," `generate_paraphrases` might produce variations like "The financial institution is closed" and "The riverbank is inaccessible." The `detect_intrinsic_ambiguity` function then calculates cosine similarity scores between the original phrase and each paraphrase. Due to differences in meaning (e.g., financial institution vs. riverbank), the scores will likely vary significantly, with the similarity range exceeding 0.2, classifying the phrase as ambiguous.

- For the phrase "He pitched the idea," paraphrases such as "He presented the idea" and "He proposed the concept" would likely show higher similarity scores, with minimal variation among them. Consequently, the similarity range would be below 0.2, classifying the phrase as unambiguous.

Upon processing, the DataFrame includes a `heuristic` column that indicates ambiguity results:

Phrase	Heuristic
"The bank is closed"	1
"He pitched the idea"	0

Table 3: Processed DataFrame with Ambiguity Detection

This code effectively identifies ambiguous phrases, making it valuable for natural language processing applications where understanding phrase clarity is crucial.

```
import pandas as pd
from transformers import T5Tokenizer, T5ForConditionalGeneration
import torch
from sklearn.metrics.pairwise import cosine_similarity

# Load T5 model and tokenizer
tokenizer = T5Tokenizer.from_pretrained('Vamsi/T5_Paraphrase_Paws')
```

```
model = T5ForConditionalGeneration.from_pretrained('Vamsi/T5_Paraphrase_Paws')

# Function to get embedding
def get_embedding(text):
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
    outputs = model.encoder(**inputs)
    return outputs.last_hidden_state.mean(dim=1).detach().numpy()

# Function to generate paraphrases
def generate_paraphrases(phrase, num_paraphrases=5, max_length=32):
    input_ids = tokenizer.encode("paraphrase: " + phrase, return_tensors="pt")
    outputs = model.generate(
        input_ids,
        max_length=max_length,
        num_beams=num_paraphrases,
        num_return_sequences=num_paraphrases,
        early_stopping=True
    )
    return [tokenizer.decode(output, skip_special_tokens=True) for output in outputs]

# Function to detect intrinsic ambiguity
def detect_intrinsic_ambiguity(phrase, num_paraphrases=5):
    paraphrases = generate_paraphrases(phrase, num_paraphrases=num_paraphrases)
    phrase_embedding = get_embedding(phrase)
    paraphrase_embeddings = [get_embedding(p) for p in paraphrases]

    scores = [
        cosine_similarity(phrase_embedding, paraphrase_embedding)[0][0]
        for paraphrase_embedding in paraphrase_embeddings
    ]

    similarity_range = max(scores) - min(scores)
    if similarity_range > 0.2:
        return 1  # ambiguous
    else:
        return 0  # not ambiguous

# Function to add ambiguity results as a new column in the DataFrame
def add_ambiguity_column(df, column_name, new_column_name="ambiguity_result",
    num_paraphrases=5):
    df[new_column_name] = df[column_name].apply(
        lambda x: detect_intrinsic_ambiguity(x, num_paraphrases)
    )

# Add ambiguity detection results to a new column 'heuristic'
add_ambiguity_column(df_sample_class_phrase, 'phrase_class', new_column_name="
    heuristic", num_paraphrases=5)

# Display the modified DataFrame
print(df_sample_class_phrase)
```

Listing 4: Python Code for Intrinsic Ambiguity Detection

Regarding the LLM approach, we utilized the remaining datasets derived from the deduction of the ample size calculated using Cochran's formula.

To construct prompts for few-shot learning, we designed the prompt with a clear structure to help the model infer the correct response patterns from minimal examples. Each prompt begins with a clear task definition, followed by several examples that illustrate the task.

Ambiguity Phrase Prompt

```
 prompt = (
f"You are an expert in linguistic analysis.  Your task is to determine if the phrase
'{phrase}' is ambiguous.
A phrase is considered ambiguous if it has multiple meanings depending on context or
interpretation.\n\n"

"Please answer **'Yes'** if the phrase has more than one distinct meaning, making
it ambiguous, "
"or **'No'** if it has a single, clear meaning, making it unambiguous.\n\n"

"Here are examples to help you understand:\n\n"

"- **Ambiguous Examples (Answer:  Yes)**:\n"
" - Phrase:  'manager years start year'\n"
" - Meaning 1:  'the year a manager started'\n"
" - Meaning 2:  'the number of years managed starting from a specific year'\n"
" - Phrase:  'note on resting place'\n"
" - Meaning 1:  'an annotation about a burial site'\n"
" - Meaning 2:  'a temporary resting location note'\n"
" - Phrase:  'Place in the Music Charts'\n"
" - Meaning 1:  'a song or album's ranking on a specific chart (e.g., Billboard)'\n"
" - Meaning 2:  'a peak or historical ranking across various charts'\n"
" - Phrase:  'Gross domestic product'\n"
" - Meaning 1:  'total economic output of a country (nominal or real)'\n"
" - Meaning 2:  'GDP measurement on a national, regional, or sector level'\n"
" - Phrase:  'original start point'\n"
" - Meaning 1:  'the origin or starting point of a journey'\n"
" - Meaning 2:  'the initial point of a project timeline or reference'\n"
" - Phrase:  'source confluence position'\n"
" - Meaning 1:  'geographic coordinates where two bodies of water meet'\n"
" - Meaning 2:  'a source document providing specific position information'\n\n"

f"Based on this information, respond with **'Yes'** if the phrase '{phrase}' is
ambiguous (multiple meanings) or **'No'** if it is not."
)
```

In Table 4, we present the percentages of detected ambiguity calculated using our two approaches for class, attribute, and relation labels. This comparison allows for an analysis of ambiguity rates across different label types, highlighting the effectiveness and variations of each approach.

	Heuristic (%)	**LLM Prompt** (%)
Class Labels	19%	15%
Attribute Labels	20%	48%
Relation Labels	18%	49%

Table 4: Percentages of Detected Ambiguity in **Phrases** by Heuristic and LLM Prompt

We observe that in Table 4, in comparison to Table 1, the percentages of detected ambiguity are consistently lower in phrases, indicating that the ontology is affected by ambiguity more significantly at the level of individual words rather than phrases. Regarding the two approaches, we note a substantial difference in their results, particularly for attribute and relation labels, highlighting variability in ambiguity detection depending on the method applied.

If additional time were available, we would manually assign labels to the remaining samples meaning the samples appeared in the Figures 8, 9 and 10 in the rectangle named "Few Shot Learning". This is particularly important for phrases, where we do not have prior knowledge of the correct answer. In contrast, with the dictionary-based approach for single words, we benefit from a predefined database that is the dictionary that provides accurate

answers. By re-running each approach with the manually labeled data, we could evaluate which method yields a more accurate and effective solution. This manual labeling process would enable a more comprehensive assessment of each approach, potentially enhancing the reliability of ambiguity detection and offering insights into the strengths and limitations of both methods.

In addition to time limitations, even with manual labeling, there is a chance that some labels may still be inaccurate, as ambiguity in phrases can sometimes be challenging to resolve. A more professional approach would involve using a referral method, where two individuals (e.g., annotators, researchers, engineers, or analysts) label or annotate the dataset independently. In cases of disagreement, a senior annotator would review and make the final decision. This method ensures a higher level of accuracy and consistency in labeling. In other words, my manual labeling could be misleading if I did it alone.

Task 2

For this assignment, we begin by reading the provided CSV file containing ESCO skill entities. We select the **preferredLabel** as it provides the primary, standardized name for each entity, enabling a more direct and accurate match with DBpedia's main identifiers. In contrast, altLabels often include synonyms and variations, which could lead to ambiguous or multiple matches, making them less suitable for precise entity mapping. Empirically, we determined that preprocessing the preferredLabel column in our dataset is necessary to avoid syntax errors in SPARQL queries. To preprocess the **preferredLabel** column for SPARQL queries, we apply several steps to clean and standardize the data, avoiding syntax errors such as:

1. **Remove Text within Parentheses**: Any text contained within parentheses is removed, such as converting "Java (computer programming)" to "Java."
2. **Remove Commas**: Removed commas such as converting "Project Manager, IT" to "Project Manager IT".
3. **Remove Apostrophes**: Any apostrophes (e.g., in "crime victims' rights") are removed to prevent them from disrupting the query.
4. **Identify Rows with Special Characters and Replace Them**: To replace specific characters with their encoded equivalents, convert # to `%23`, + to `%2B`, ; to `%3B`, and / to `%2F`.

Through these steps, we prepare the **preferredLabel** column for compatibility with SPARQL by eliminating or flagging problematic characters, thus reducing the risk of syntax errors. Of course, these transformations may not be completely accurate, as another developer might decide to handle them differently. For example, someone could say to remove all lines having syntax errors (like drop "na" strategy) or fill them with the correct value, but the correct value sometimes is relative (so at this point there is space for "confusion"). To further improve matching accuracy, we generate multiple DBpedia-compatible URI variants for each **preferredLabel**. These variants include different formatting options, such as lowercase with underscores, title-cased, and combinations thereof, increasing the likelihood of aligning with DBpedia's naming conventions. For example, for the label **Data Science**, the following DBpedia-compatible URI variants are generated to maximize matching accuracy:

- `data_science`
- `Data_science`
- `data_Science`
- `Data_Science`

These variations combine lowercase and capitalized forms with underscores, aligning with common DBpedia naming conventions. In Figure 5 is presented the related code. Next, we initiate a SPARQL query (limiting the result for the English language as the given CSV is named "en," implying English only) for each variant to retrieve relevant class and label information. For each query, we employ pagination to ensure complete retrieval of results, filtering out empty or irrelevant results. Only variants with valid matches are retained, enabling a precise alignment with DBpedia entities. By utilizing parallel processing, we expedite the querying process, executing multiple queries simultaneously. This approach significantly enhances efficiency, particularly given the potential size of the dataset. In Figure 6 is the relative code. The final output is an updated DataFrame, containing the matched DBpedia classes, labels, and only those URI variants that returned valid results, providing a structured and comprehensive mapping between ESCO skills and DBpedia classes.

```python
def format_to_specific_dbpedia_variants(label):
    """
    Convert a label to multiple DBpedia URI format variants:
    - all lowercase with underscores
    - first word capitalized, others lowercase
    - first word lowercase, each subsequent word capitalized
    - all words capitalized with underscores
    """
    words = label.split()
    formatted_variants = [
        "_".join(word_format) for word_format in product(*[
            [word.lower(), word.capitalize()] for word in words
        ])
    ]
    return list(set(formatted_variants))  # Remove duplicates
```

Listing 5: Python Code for the creation of DBPedia variants

```python
def query_dbpedia_for_variant(variant):
    """
    Perform the SPARQL query for a single variant to retrieve classes and labels.
    Returns classes_labels list only if there are results.
    """
    unique_classes = set()
    classes_labels = []
    offset = 0

    while True:
        query = f"""
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX dbr: <http://dbpedia.org/resource/>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

        SELECT ?class ?label
        WHERE {{
          dbr:{variant} rdf:type ?class .
          ?class rdfs:label ?label .
          FILTER (lang(?label) = 'en')
        }}
        LIMIT 100 OFFSET {offset}
        """
        sparql.setQuery(query)
        sparql.setReturnFormat(JSON)

        try:
            results = sparql.query().convert()
            # Stop if there are no more results
            if not results['results']['bindings']:
                break

            for result in results['results']['bindings']:
                class_uri = result['class']['value']
                label = result['label']['value']
                if class_uri not in unique_classes:
                    unique_classes.add(class_uri)
                    classes_labels.append((variant, class_uri, label))  # Include
                        variant in the result

            offset += 100  # Increment offset for pagination

        except Exception as e:
            print(f"Error querying {variant}: {e}")
            break
```

```
    # Only return the variant if there are results
    return classes_labels if classes_labels else None


def get_all_classes_and_labels_for_variants(variants):
    """
    Run SPARQL queries for all variants in parallel and gather unique classes and
        labels.
    """
    classes_labels = []
    valid_variants = set()  # Track only the valid variants with results

    # Use ThreadPoolExecutor for multithreading
    with ThreadPoolExecutor(max_workers=10) as executor:
        # Submit all queries to be executed in parallel
        future_to_variant = {executor.submit(query_dbpedia_for_variant, variant):
            variant for variant in variants}

        # Process results as they complete
        for future in tqdm(as_completed(future_to_variant), total=len(variants), desc
            ="Processing Variants"):
            variant_classes_labels = future.result()
            if variant_classes_labels:
                classes_labels.extend(variant_classes_labels)
                # Collect valid variants only when there are results
                valid_variants.update([v[0] for v in variant_classes_labels])

    # Separate classes and labels into two lists for the DataFrame
    class_uris = [cl[1] for cl in classes_labels]
    labels = [cl[2] for cl in classes_labels]

    return class_uris, labels, list(valid_variants)  # Include only valid variants
        with results
```

Listing 6: Python Code for Retrive DBPedia Data and Run the process in parallel mode.

Using the DataFrame containing integrated information from the ESCO CSV file and the DBpedia ontology, we applied a filter to view the mapped elements, specifically focusing on ESCO entities that have been aligned with DBpedia entities belonging to recognized classes within the DBpedia ontology. Despite the variants of names of entities, approximately 20% of the entities in ESCO are mapped to DBpedia entities.

Regarding the structure of our DataFrame containing the integrated data, we have the following columns:

- **preferred_label_esco**: This column contains the primary label or name of the entity as defined in ESCO, providing a standardized term for each entity.
  Example: "public health"
- **potential_entity_variants_dbpedia**: This column lists possible variants of the entity name as may be found in DBpedia, capturing different forms or spellings to account for variations in DBpedia.
  Example: ["Public_health", "Public_Health", "public_Health", "public health"]
- **class_label_dbpedia**: This column shows the class or category labels assigned to the entity within the DBpedia ontology, representing its classification or type.
  Example: ["person function", "person function"]
- **entity_valid_variants_dbpedia**: This column contains validated variants of the entity names that are specifically recognized in DBpedia. These are the accepted forms of the entity that align with the DBpedia ontology.
  Example: ["Public_health", "Public_Health"]

Note: If the lists in columns `class_label_dbpedia` and `entity_valid_variants_dbpedia` are empty, it indicates that the ESCO entities did not match any corresponding entities in DBpedia.

Then, to facilitate our LLM prompt and analysis, we expanded our DataFrame both in rows and columns. We observed that the column `class_label_dbpedia` may contain multiple values. To enable line-by-line processing

of entity-class pairs in our language model (LLM), we separated each value into individual rows. In this expanded format, each entity-class pair occupies a single row, while the other values in the DataFrame remain constant. The individual values generated from this process were saved in a new column named `llm_class_label`. Additionally, we introduced another column named `llm_formatted_entity`, which contains individual values of the entity name, essentially duplicating the values from the `preferred_label_esco` column. This restructuring allows each entity-class pair to be processed independently while maintaining clarity and consistency in our data analysis. As a result of this expansion process, the DataFrame's shape changed from (**564**) rows to (**910**) rows.

The next main task after data integration is the LLM Prompt task. To create a professional prompt utilizing the **few-shot learning** technique, we first took a random sample of 10 rows from the DataFrame to use as examples for generating expected results. These 10 rows were then removed from the main dataset to prevent any bias in the results.

Additionally, we added some custom examples to balance the dataset with pairs of entities and classes that should be assigned as true (indicating the entity belongs to the class) and others as false (indicating the entity does not belong to the class). We ensured that these added examples were unique and did not exist in our dataset to maintain the integrity of the data. In the box below is presented our prompt.

After running our LLM prompt on the DataFrame and performing an analysis, we observed that only 86 rows out of 910 were assigned a value of **1** by the LLM, **indicating that the entity belongs to the specified class**. Through a manual check, we confirmed that the LLM's classifications were accurate for nearly all these rows. The dominant classes were **software**, **language**, and **medical specialty**. We could say that **concrete objects**, such as the entity `tomcat` for the class `Software Product` , may be easier to categorize properly in comparison to non-concrete objects (meaning abstract objects). For the elements that LLM assigned with 1, in Tables 5, 6 in the column `Entity label`, the majority of terms are concrete, referring to identifiable software (products), languages, companies or specific medical specialties. More specifically, in our samples, we have detected the following concrete objects:

- **Software Products**:
  - `xcode`, `unreal engine`, `moodle`, `apache tomcat`, `adobe photoshop`, `joomla`, `drupal`, `adobe illustrator`, `wireshark`, `windows phone`, `oracle warehouse builder`, `microsoft visio`, `microsoft access`, `apache maven`, `capture one`, `kali linux`, `taleo`
- **Companies**:
  - `cisco`, `edmodo`
- **Minerals and Chemical Substances**:
  - `graphite`, `salt`, `shale gas`
- **Cultural or Mythological Entities**:
  - `shiva` (a deity, often considered a specific, identifiable figure in cultural and religious contexts)

So, based on the (entity, class) pairs assigned a value of 1 (indicating that the entity belong to the class), we can infer that concrete objects are more likely to be assigned correctly to their respective classes. **This observation suggests a higher accuracy in class assignment for concrete entities compared to abstract ones**.

Prompt detecting if pairs (entity class) is correct.

In the DBpedia ontology, determine if the **entity** `'entity'` belongs to the **class** `'class_name'`. DBpedia categorizes entities into various classes like `'organisation'`, `'language'`, `'person'`, `'book'`, `'music genre'`, etc.

Please respond with **'Yes'** if the entity is an instance of the specified class and **'No'** if it is not.

Use the following examples as a guide:

- **Examples where the answer is 'No':**
- Entity: 'acoustical engineering', Class: 'organisation' → Answer: No
- Entity: 'petrology', Class: 'organisation' → Answer: No
- Entity: 'forensic psychiatry', Class: 'person function' → Answer: No
- Entity: 'soil mechanics', Class: 'organisation' → Answer: No
- Entity: 'fish anatomy', Class: 'book' → Answer: No
- Entity: 'special needs education', Class: 'music genre' → Answer: No
- Entity: 'in-circuit test', Class: 'building' → Answer: No
- Entity: 'bioethics', Class: 'book' → Answer: No

- **Examples where the answer is 'Yes':**
- Entity: 'prolog', Class: 'language' → Answer: Yes
- Entity: 'prolog', Class: 'programming language' → Answer: Yes
- Entity: 'ios', Class: 'place' → Answer: Yes
- Entity: 'ios', Class: 'populated place' → Answer: Yes
- Entity: 'data scientist', Class: 'person' → Answer: Yes
- Entity: 'school', Class: 'organisation' → Answer: Yes
- Entity: 'bank', Class: 'organisation' → Answer: Yes
- Entity: 'Pop', Class: 'music genre' → Answer: Yes

Based on these examples, provide your answer as 'Yes' or 'No' only.

Entity Label	Class Label
geriatrics	medical specialty
otorhinolaryngology	medical specialty
yiddish	language
matlab	programming language
autism	disease
coffeescript	programming language
php	programming language
sanskrit	language
xcode	software
unreal engine	software
hindi	language
moodle	software
ruby	programming language
sexology	book
apache tomcat	software
latin	language
cued speech	language
neurology	medical specialty
horse riding	sport
golf	activity

Table 5: Entity-Class Pairs (1)

Entity Label	Class Label
rheumatology	medical specialty
cardiology	medical specialty
adobe photoshop	software
arabic	language
cisco	company
dermatology	medical specialty
taleo	company
occitan	language
haskell	programming language
synfig	software
perl	programming language
capture one	software
endocrinology	medical specialty
salt	mineral
joomla	software
urdu	language
wireshark	software
ophthalmology	medical specialty
windows phone	software
emergency medicine	medical specialty

Table 6: Entity-Class Pairs (2)

Below is a detailed analysis. In order to follow up with the initial question—namely, which DBpedia classes tend to have the most mistakes—we removed the rows in the DataFrame that were assigned a value of 1, indicating that

the entity belongs to the specified class. This allowed us to focus on analyzing the remaining rows for answering our intial question. So, we left with 809 rows. Applying some simple mathematical computations, we derived the plot presented in Figure 11, assuming that the LLM reflects reality—meaning that the label assigned as "not belonging" (being the binary value 0) indicates the entity does not fit the current class.

In Figure 11, we depict the number of classes that appear to have the most mistakes. At the top are the classes **person function**, **music genre**, and **organisation**.

Note: In this diagram 11, we have included only the names of classes having at least 10 data rows. In category other, we have assigned all classes having less than 10 data rows.

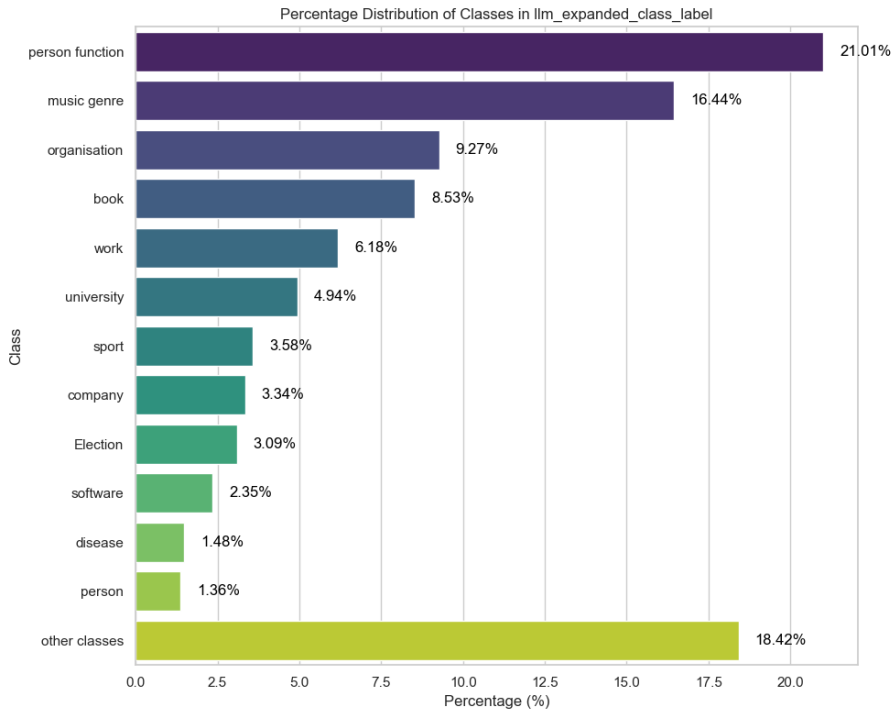


Figure 11: Distribution of Incorrect Class Assignments by Percentage

Due to time limitations, we dived into the **top three categories meaning classes**, and we took a sample of pairs (entity, class) to examine whether the terms are concrete or **abstract**. The following tables, 7 and 8 being related to the class person function, list terms from the `entity label` column, each of which is classified as an abstract concept. These terms represent fields of study, professions, or practices, making them conceptual rather than concrete, physical objects. Similarly, tables 9 and 10 being related to the class music genre, include terms from the `entity label` column that are also abstract concepts. These terms encompass fields of study or knowledge, philosophical domains, technological and analytical practices, medical and scientific fields, and broad categories (e.g., wildlife). Likewise, tables 11 and 12 being related to the class organisation, present terms from the `entity label` column that are classified as abstract concepts, representing fields of study, legal disciplines, and skills or personal qualities.

It is important to note that in Tables 9, 10, 11, 12, 7, and 8, the majority of elements—primarily **abstract terms**—have been assigned to incorrect class names (see the *Class Label* column). In contrast, Tables 5 and 6, which contain entities that include some **concrete terms**, show correct assignments to their respective classes. **Therefore, based on DBPedia ontology, it appears valid to accept the assumption in our task that abstract concepts are generally more challenging to model accurately than concrete ones**.

Note: As previously mentioned, due to time constraints, we focused on samples from the three classes with the highest rates of incorrect assignments. Given additional time, a more rigorous approach would have been to select a sample using the Cochran formula, which would yield results that are scientifically acceptable and generalizable. However, this approach would require analyzing a considerably larger sample size, which led us to reject it.

Entity Label	Class Label
vascular surgery	person function
natural history	person function
optics	person function
animal training	person function
epigraphy	person function
food engineering	person function
neonatology	person function
carpentry	person function
ophthalmology	person function
electroencephalography	person function

Table 7: Entity-Class:**person function** Pairs (1)

Entity Label	Class Label
monasticism	person function
cyber security	person function
typography	person function
musical instruments	person function
family law	person function
grammar	person function
pedagogy	person function
agronomy	person function
reiki	person function
pharmaceutical industry	person function

Table 8: Entity-Class:**person function** Pairs (2)

Entity Label	Class Label
occultism	music genre
database	music genre
data protection	music genre
desktop publishing	music genre
machine translation	music genre
morality	music genre
child psychiatry	music genre
web analytics	music genre
wildlife	music genre
ethics	music genre

Table 9: Entity-Class:**music genre** Pairs (1)

Entity Label	Class Label
geography	music genre
molecular biology	music genre
reverse engineering	music genre
logic	music genre
data protection	music genre
ajax framework	music genre
metaphysics	music genre
triplestore	music genre
biostatistics	music genre
literature	music genre

Table 10: Entity-Class:**music genre** Pairs (2)

Entity Label	Class Label
cisco	organisation
computer forensics	organisation
neurophysiology	organisation
embryology	organisation
commercial law	organisation
neurology	organisation
attention to detail	organisation
environmental engineering	organisation
pharmacokinetics	organisation
occupational medicine	organisation

Table 11: Entity-Class:**organisation** Pairs (1)

Entity Label	Class Label
ophthalmology	organisation
chemistry	organisation
audiology	organisation
cardiology	organisation
immunology	organisation
edmodo	organisation
ornithology	organisation
hydrography	organisation
trigonometry	organisation
mycology	organisation

Table 12: Entity-Class:**organisation** Pairs (2)

Acknowledgment

I would like to express my sincere gratitude to Professor Panos Alexopoulos for his invaluable guidance and support throughout the course *Knowledge Graphs with Large Language Models*. His expertise and dedication have greatly enhanced my understanding of the subject, and his encouragement has been instrumental in the development of this work.

I am deeply appreciative of the opportunity to learn under his course and am genuinely interested in potential future collaborations or discussions that could build upon the foundation laid in this course. Thank you, Professor Alexopoulos, for inspiring my passion for this field and for the lasting impact of your teaching.