

Chapter 5

Overview of the Implementation of the Local Holistic KGC Pipeline

The main outcome of this research is a functional KGC pipeline prototype that is capable of converting a semi-structured academic paper into its KG representation. Therefore, in this section and the following Sections 6, 7, 8, 9 10, we will take a close look at the pipeline and discuss the working principle behind it. Meanwhile, we will also discuss the key decisions made during the implementation of the pipeline.

5.1 Interface

- **Input:** The semi-structured representation of an academic paper (RDF-Based Metadata Graph) in TTL or our customised JSON format.
- **Output:** The Knowledge Graph representation of the paper (RDF-Based Knowledge Graph) in TTL or our customised JSON format.
- **Language:** Python 3.12.4.
- **Software Platform:** Linux, MacOS.
- **Hardware Platform:** CUDA, Metal.
- **LLM Used:** Meta-Llama-3-8B-Instruct-Q4-0, bge-english-base, bge-m3.
- **Key Packages Used:** GPT4ALL (loading LLM decoders), FlagEmbedding (loading LLM encoders), NLTK (providing basic natural language processing functionality like lemmatisation).
- **Availability:** <https://gitlab.anu.edu.au/u1085404/anu-scholarly-kg/-/tree/master/llm-based/LLM-KGC-v0>

5 Overview of the Implementation of the Local Holistic KGC Pipeline

The summary of the interface is shown as above. Note that:

- GPT4ALL (GPT for All) and GPT-4 are two distinct things. GPT4ALL provides a platform to load many LLM decoders into Python. It also enables the Chat Completion functionality of a decoder model ([Nomic, 2024](#)), i.e., the decoder model loaded by GPT4ALL will answer a user's prompt, instead of generating anything that continues the prompt, as previously discussed in Section [2.4.4](#).
- Although RDF-Graph is commonly stored in the format of TTL, as discussed in Section [2.2.2](#), we have chosen to use JSON to store the KG for some special reasons. In fact, our pipeline will *first convert a TTL-format KG into our customised JSON-format KG* and work on this customised JSON format through the KGC stages because a JSON file can be loaded into Python as a Python Dictionary. In contrast, the access of TTL files relies on SPARQL Query, which is not natively supported by Python. More will be discussed in the next Section [5.4](#).

5.1.1 LLM Encoders and Decoders Used

Decoder 1:

- **Named:** Meta-Llama-3-8B-Instruct.Q4_0.gguf
- **Size:** 4.66GB
- **Context Limit:** 8192
- **Availability:** <https://huggingface.co/QuantFactory/Meta-Llama-3-8B-Instruct-GGUF/tree/main>

Encoder 1:

- **Named:** BAAI--bge-base-en-v1.5
- **Size:** 438.9MB
- **Context Limit:** 512
- **Availability:** <https://huggingface.co/BAAI/bge-base-en>

Encoder 2:

- **Named:** BAAI--bge-m3
- **Size:** 2.29GB
- **Context Limit:** 8192
- **Availability:** <https://huggingface.co/BAAI/bge-m3>

In the later sections and chapter, especially when presenting an algorithm, we will refer to them as LLaMA, BGE, and M3, respectively.

5.1.2 Development Set Used

Paper: “J2RM: an Ontology-based JSON-to-RDF Mapping Tool” (Rodríguez Méndez et al., 2020) Availability: <https://ceur-ws.org/Vol-2721/paper593.pdf>

We have chosen a single academic paper from ASKG as our primary development set (validation set). Although, there is only one paper. The paper itself consists of a number of sections, paragraphs, and sentences, which is enough to cover relatively diverse linguistic patterns and help us observe different edge cases during our NLP tasks. It can also accelerate our development, but the drawback is that it can still potentially make the implementation of our pipeline biased to the specific subset of linguistic patterns contained in the paper.

We have chosen SciERC (Luan et al., 2018) and DocRED (Yao et al., 2019) as our secondary datasets. These datasets both provide mappings between documents and KGs. However, neither is considered to contain a full article. SciERC’s sources come from academic papers, but every single sample contains the abstract only. DocRED’s sources mainly come from Wikipedia, but they are also shortened, containing only a single paragraph. Therefore, the high performance of these datasets does not necessarily mean that our pipeline performs well.

Furthermore, the ground truth KGs from these two datasets are considered as KGs based on traditional KGC according to our literature review. This is because the target KGs of both datasets only consist of a limited number of predefined entity types and predicates. For example, SciREC only contains predefined six academic entity types, like Task and Method, and seven academic predicates, like Use-for. DocRDF contains six general entity types, like Person and Location and 96 predicates. Therefore, if we only aim to achieve a high benchmark score on these two datasets, we essentially confine our pipeline to produce traditional KG, instead of the KG based on the idea of open information extraction. However, they are still useful in some specific cases, and we have used them during our development.

Both our primary and secondary datasets will be mentioned in Section 6 to Section 10 to demonstrate some decisions made during the development of our pipeline, especially our prompts. This can be compared to hyper-parameter turning in traditional machine learning, but we are turning the prompts in this study.

Next, we will go through both the high-level data structure and low-level data structure of our input and output. This is highly relevant to our algorithms, which will be discussed in the later sections.

5.1.3 High-Level Input and Output Data Structure

The high-level input and output are shown in 5.1. This is slightly more detailed than the one shown in the introduction. It is shown here for comparison with other low-level inputs and outputs. Note that we have eliminated some intermediate fields stored in

5 Overview of the Implementation of the Local Holistic KGC Pipeline

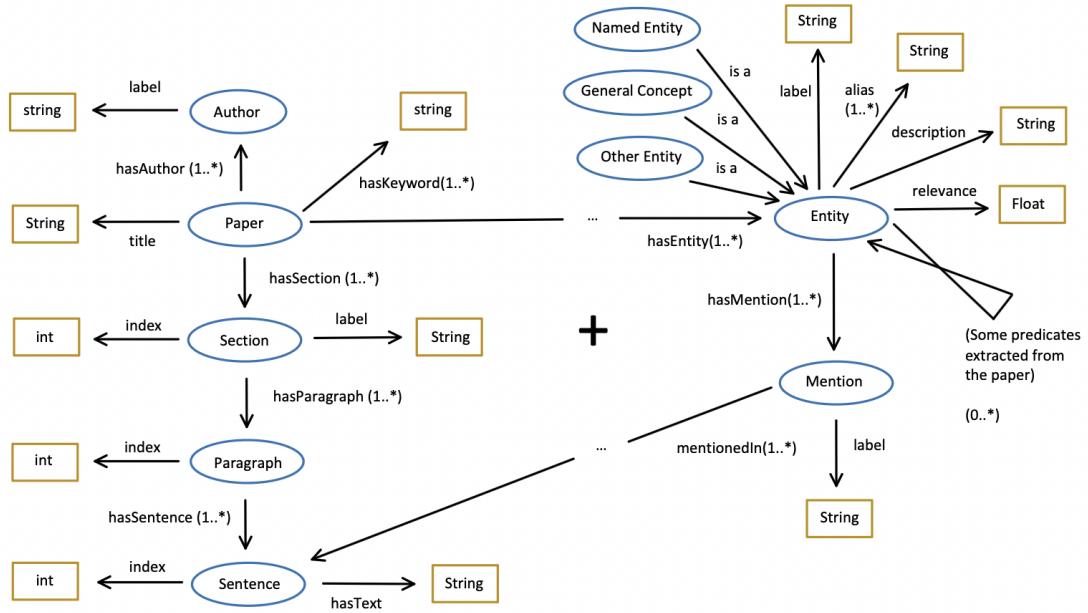


Figure 5.1: High-Level Input and Output

this data structure. They are important during the processing but are deleted when the pipeline finishes.

5.1.4 Low-Level Input (TTL and JSON)

TTL Pseudo-Code

```

paper_1_iri      is_a          Paper ;
                  has_title     "Research on the History of ANU" ;
                  has_author    "Amy",
                  has_author    "Ben" ;
                  has_keyword   "ANU",
                  has_keyword   "History";
                  has_section   section_1_iri,
                  has_section   section_2_iri,
                  has_section   section_3_iri .
...

section_2_iri    is_a          Section ;
                  has_index     2 ;
                  has_label     "Introduction" ;
                  has_paragraph paragraph_1_iri,
                  has_paragraph paragraph_2_iri,
                  has_paragraph paragraph_3_iri .

paragraph_1_iri  is_a          Paragraph ;
                  has_index     1 ;
                  has_sentence  sentence_1_iri,

```

```

    sentence_2_iri,
    sentence_3_iri .

...
sentence_1_iri      is_a          Sentence ;
                    has_index     1 ;
                    has_text      "ANU is a university located in Canberra.".
...

```

JSON Pseudo-Code

```

{
  "iri": "paper_1_iri",
  "title": "Research on the History of ANU",
  "authors": [
    "Amy",
    "Ben"
  ],
  "keywords": [
    "ANU",
    "History"
  ],
  "sections": [
    {...},
    {
      "iri": "section_2_iri",
      "label": "Introduction",
      "paragraphs": [
        {
          "iri": "paragraph_1_iri",
          "sentences": [
            {
              "iri": "sentence_1_iri",
              "text": "ANU is a university located in Canberra."
            },
            {
              "iri": "sentence_2_iri",
              "text": "It has a long history."
            },
            {...}
          ]
        },
        {...},
        {...}
      ]
    },
    {...}
  ]
}

```

Up to now, it should be clear why we prefer JSON to TTL. The original graph data structure, RDF-Graph, has no sense of order and hierarchy. Everything is just a triple sitting on a plane. The hierarchy has to be inferred based on the relations of nodes, and the orders of sections, paragraphs, and sentences have to be stored in extra index nodes. However, the semantics of the semi-structured representation of a paper is hierarchical and ordered.

5 Overview of the Implementation of the Local Holistic KGC Pipeline

In contrast, JSON is hierarchical, and its list is ordered. When JSON is implemented by Python (> 3.7), i.e., it is parsed by Python into Python Dictionary or a Python Dictionary is dumped into JSON, *its key is also ordered*. That is, `iri` and `title` are stored in a list instead of a set. This is why JSON (disk-level) or Python Dictionary (memory-level) is chosen. Note that an RDF-Graph or RDF-KG is a high-level concept. How such a graph is stored at a low level can vary.

5.1.5 Low-Level Output (TTL and JSON)

TTL Pseudo-Code

```
/* All triples in the input are preserved plus: */

entity_1_iri      is_a          Named_Entity ;
                   has_label      "The Australian National University" ;
                   has_alias      "ANU",
                   has_description      "The Australian National University",
                   has_relevance_score  "Australian National University" ;
                   has_mention     "...";
                   predicate_1_iri   entity_2_iri;

entity_2_iri      is_a          Named_Entity ;
                   has_label      "Canberra" ;
                   has_alias      "Canberra",
                   has_description      "CBR",
                   has_relevance_score  "...";
                   has_mention     entity_2_mention_1_iri,
                   ... .

entity_1_mention_1_iri  is_a          Mention ;
                        has_label      "ANU" ;
                        mentioned_in    sentence_1_iri .

entity_2_mention_1_iri  is_a          Mention ;
                        has_label      "Canberra" ;
                        mentioned_in    sentence_1_iri .

entity_1_mention_1_iri  is_a          Mention ;
                        has_label      "ANU" ;
                        mentioned_in    sentence_1_iri .

predicate_1_iri      is_a          Predicate ;
                   has_label      "located in" ;
                   has_description  "...".

Named_Entity         sub_class_of  Entity .
General_Concept       sub_class_of  Entity .
Other_Entity          sub_class_of  Entity .
```

JSON Pseudo-Code

```
{
  /* All fields in the input are preserved plus */
  "nodes":{
```

```

"entity_1_iri": {
    "label": "The Australian National University",
    "aliases": [
        "ANU",
        "The Australian National University",
        "Australian National University"
    ],
    "node_type": "Named Entity",
    "description": "...",
    "relevance_score": 0.9,
    "mentions": {
        "entity_1_mention_1_iri": {
            "local_name": "ANU",
            "reference": "sentence_1_iri"
        },
        ...
    }
},
"entity_2_iri": {
    "label": "Canberra",
    "aliases": [
        "Canberra",
        "CBR"
    ],
    "node_type": "Named Entity",
    "description": "...",
    "relevance_score": 0.8,
    "mentions": {
        "entity_2_mention_1_iri": {
            "local_name": "Canberra",
            "reference": "sentence_1_iri"
        },
        ...
    }
},
...
},
"edges": {
    "predicate_1_iri": {
        "label": "located in",
        "description": "..."
    },
    ...
},
"triples": [
    [
        "entity_1_iri",
        "predicate_1_iri",
        "entity_2_iri"
    ],
    ...
]
}

```

The key difference is that, instead of storing everything in the KG as triples, like TTL. We have stored it using a hybrid approach.

- For the relations that only belong to an entity, e.g., aliases, descriptions, and mentions, they are stored directly under the Entity's local fields instead of the global field "triples".

5 Overview of the Implementation of the Local Holistic KGC Pipeline

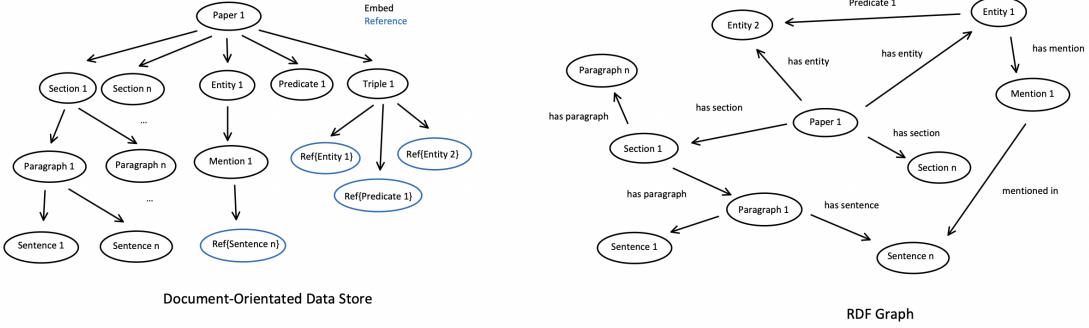


Figure 5.2: Original Input and Modified Input

- For the relations between any two entities, they are stored in the global fields of the fields, like like in TTL. Therefore, our overall KG in JSON is stored as a list of nodes, plus a list of edges, plus a list of triples.

In this way, we can easily observe any local relations about the entities, e.g, their labels, and aliases, by checking the entity itself and any global relations, i.e., the relations between any two entities, in a single `triples` list. This makes it easy for us to directly observe and debug the relations between entities in the raw JSON file without the assistance of any KG viewing tool. Whereas in TTL,

In TTL, however, relations scatter among each entity to which the subject of the relation belongs, as shown in the TTL pseudo-code. In fact, in TTL, there is even no guarantee that the triple `(entity_1_iri, predicate_1_iri, entity_2_iri)` is under the entity `entity_1_iri`. It can be placed anywhere in the file, and there is a sense of ordering or hierarchy of triples. Therefore a graphical tool is usually needed to observe a KG in TTL format.

Our pipeline converts our intermediate JSON-base KG back to TLL as our output because, of course, TTL is the most common and widely supported way to store a KG.

Table 5.1 and Figure 5.2 provides a different view of the input and output of the pipeline. Essentially, our JSON-Based modified input can be regarded as a Document-Orientated Data Store, which maintains some hierarchical structure of the document as well as storing triples and representing semantics.

5.2 Pipeline Structure

Figures 5.3 and 5.4 collectively show the overall structure of our pipeline. The box coloured green indicates that this sub-stage involves an LLM decoder, the box coloured blue indicates LLM encoders are involved, and the box coloured black indicates the stage is purely algorithmic. The following sub-section will briefly go though each stage and

Table 5.1: Different View of Input and Output

Semantical	A Semi-Structured Paper	Intermediate KG	A KG of the Paper
Logical	RDF-Graph	Document-Orientated Data Store *	RDF-Graph
Disk	TTL	JSON	TTL
Memory	RDF Graph Object	Python Dictionary	RDF Graph Object

demonstrate how an input JSON/TTL Metadata Graph of a paper is converted into a JSON/TTL Knowledge Graph. The boxes placed in parallel means the substages are not dependent on each other and are ideally run in parallel.

5.2.1 Stage 0: Data Pre-Processing

The first stage involves converting TTL-Based KG into our customised JSON format so that when a KG is loaded into memory, it can become a natively-supported Python Dictionary for our later algorithm to process. Since our pipeline is primarily designed for taking semi-structured papers from ASKG as inputs, most of the time, our inputs are in TTL.

Another key sub-stage is the integrity and consistency check in order to prevent our later algorithms from failing. As mentioned in Section 2.2.3, the RDF Graph is designed to be highly flexible, and hence there are usually no constraints explicitly coded, unlike the relational model. For example, there is no guarantee that a Paper node has at least one section, even though we have stated such constraints in our ontology graph (Figure 5.1). In fact, when we first receive our input data from ASKG, there are indeed some integrity and consistency issues. For example, we have found that some papers have two titles and some papers have sentences or sections missing. These papers need rejection and reporting.

Therefore, this sub-stage checks the following:

- A paper has *exactly* one title.
- A paper has *at least* one author, keyword, and section.
- A section has *exactly* one subtitle.
- A section has *at least* one paragraph.
- A paragraph has *at least* one sentence.
- A sentence has *at least* one piece of text.

5.2.2 Stage 1: Entity Extraction, Typing, and Classification

As shown in Figure 5.3, this stage involves extracting all entity mentions in the text, assigning them one or more potential parent types, and classifying them into either

5 Overview of the Implementation of the Local Holistic KGC Pipeline

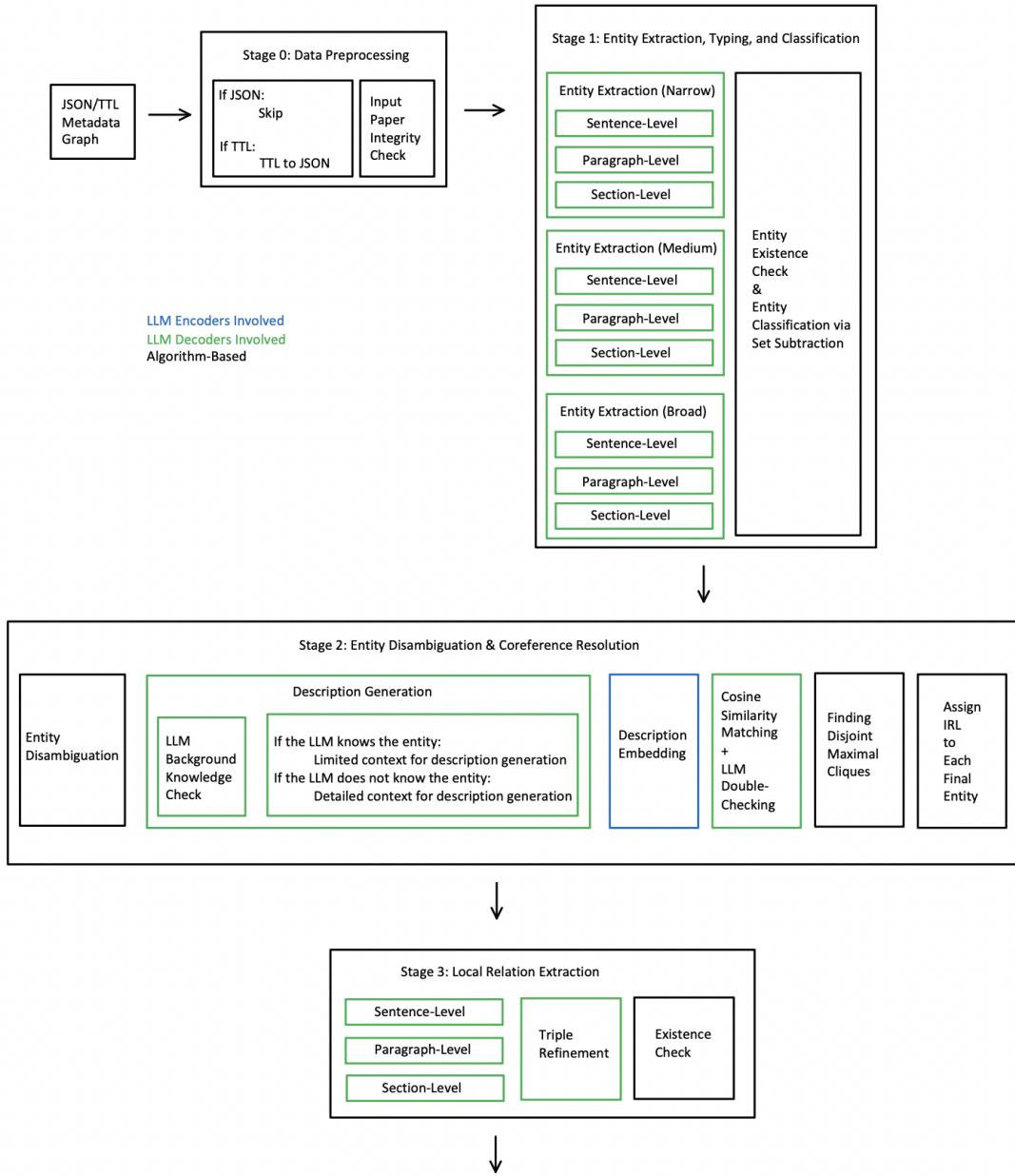


Figure 5.3: Pipeline Structure: Stage 0 - Stage 3

5.2 Pipeline Structure

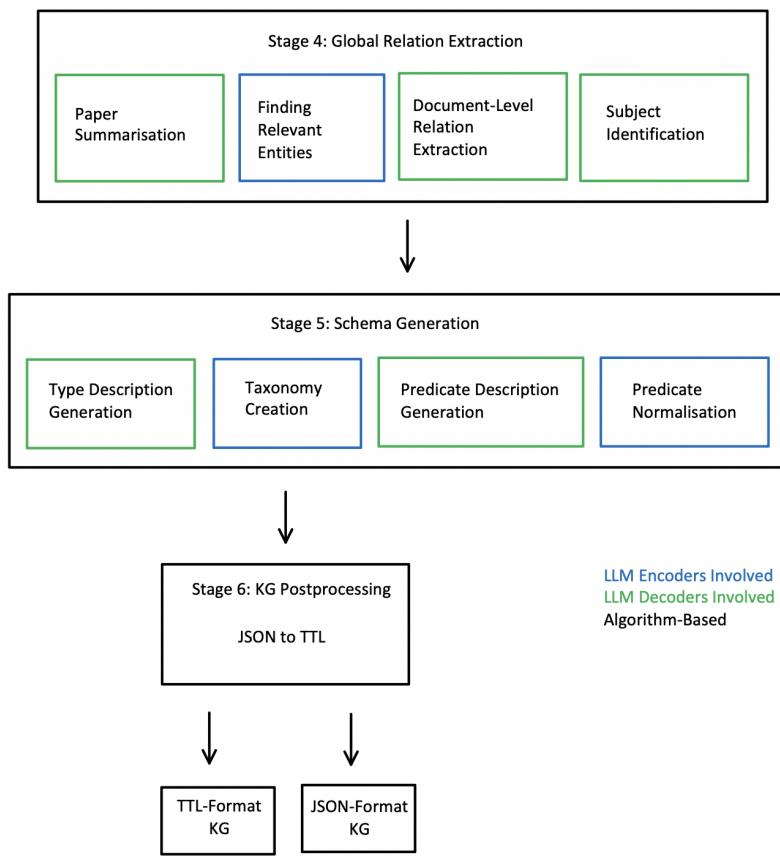


Figure 5.4: Pipeline Structure: Stage 3 - Stage 6

5 Overview of the Implementation of the Local Holistic KGC Pipeline

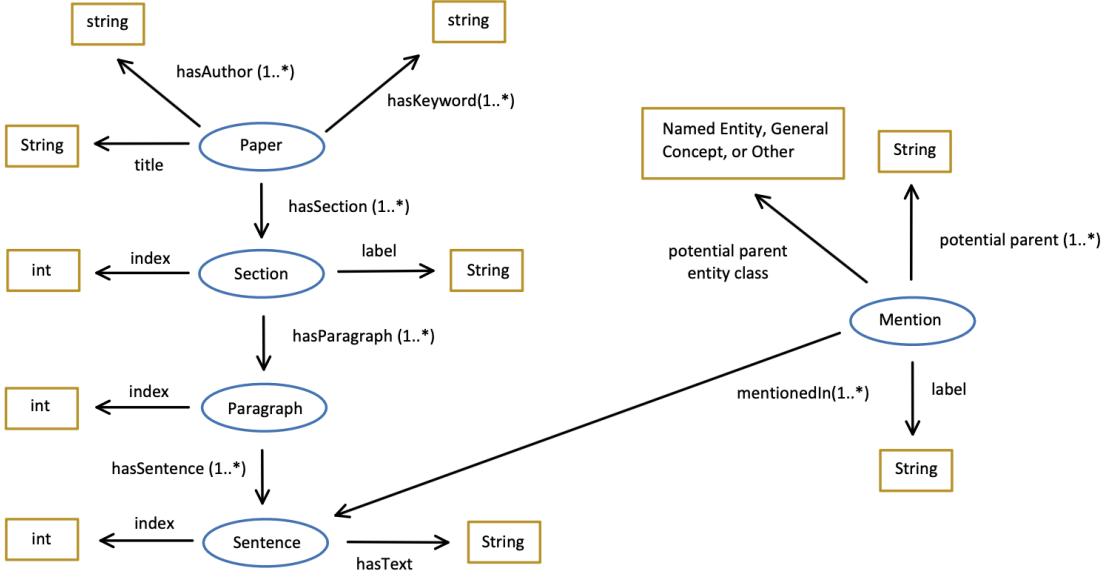


Figure 5.5: Intermediate KG Ontology after Stage 1

Named Entity, General Concept, or Other.

The extraction process can be divided into three sub-stages: narrow, medium, and broad. This is useful when we perform entity classification via set subtraction. The narrow entity extraction aims to extract only the named entity in the document. The medium entity extraction aims to extract both the named entity and the general concept. The broad entity extraction aims to extract all entities. Therefore, by set subtraction, we can classify our extracted entities into Named Entity, General Concept, and Other.

The extraction processes also go from sentence level (looping through every sentence) up to section level (looping through every section), i.e., fetching the LLM sentence by sentence or section by section to extract entities.

After the extraction, our pipeline checks if the entity is present in the document for each entity. The reasons for doing all these sub-steps will be discussed in Section 6.

After Stage 1, all Mention nodes will be created in our KG. Figure 5.5 shows the intermediate KG ontology after stage 1. Note that the entity nodes have not yet been created until Stage 2. Note that we use the term “type” to refer to a parent of an entity. For example, “ANU” has the type “University”. We use the term “class” to refer to our broad classification of entity, e.g., “ANU” has the class “Named Entity”.

5.2.3 Stage 2: Entity Disambiguation and Coreference Resolution

This stage, as shown in Figure 5.3, involves linking all mentions referring to the same thing together into Entity nodes. The ontology of the intermediate KG after this stage

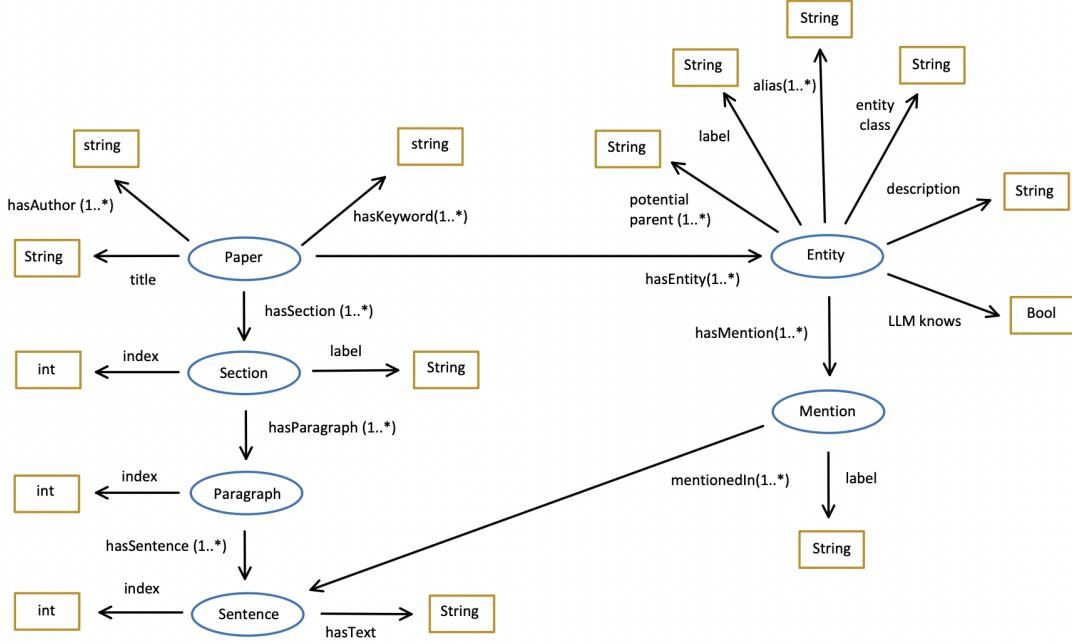


Figure 5.6: Intermediate KG Ontology after Stage 2

is shown in Figure 5.6. At the beginning of this stage, we assume that all Mentions refer to distinct entities. Therefore, we create an Entity node for each Mention.

We first perform Entity Disambiguation to merge entities with the same label (name).

Next, we prompt the LLM decoder to generate a description for each entity, whether how a description is generated is based on whether the LLM has background knowledge of this entity. This will ensure a higher-quality description.

Then, we use the description, the entity label, and the entity types to generate the embedding for each entity using the LLM encoder.

If it is found that two embeddings are similar, we prompt the LLM decoder to perform a double-check. If it is found that two entities are similar, we draw a temporary edge between these two entities.

We will merge a group of entities into a single entity only if they form a complete graph in our intermediate KG (cliques).

When merging any two entities into one entity, the mentions of the new entity are then the union of mentions of the two entities, similar to potential parent types and aliases. The label, i.e., the representative name of the entity, is chosen to be the label of the mention that first appears in the document, similar to the description. The other names that are not chosen are stored as aliases. This is also due to the assumption that

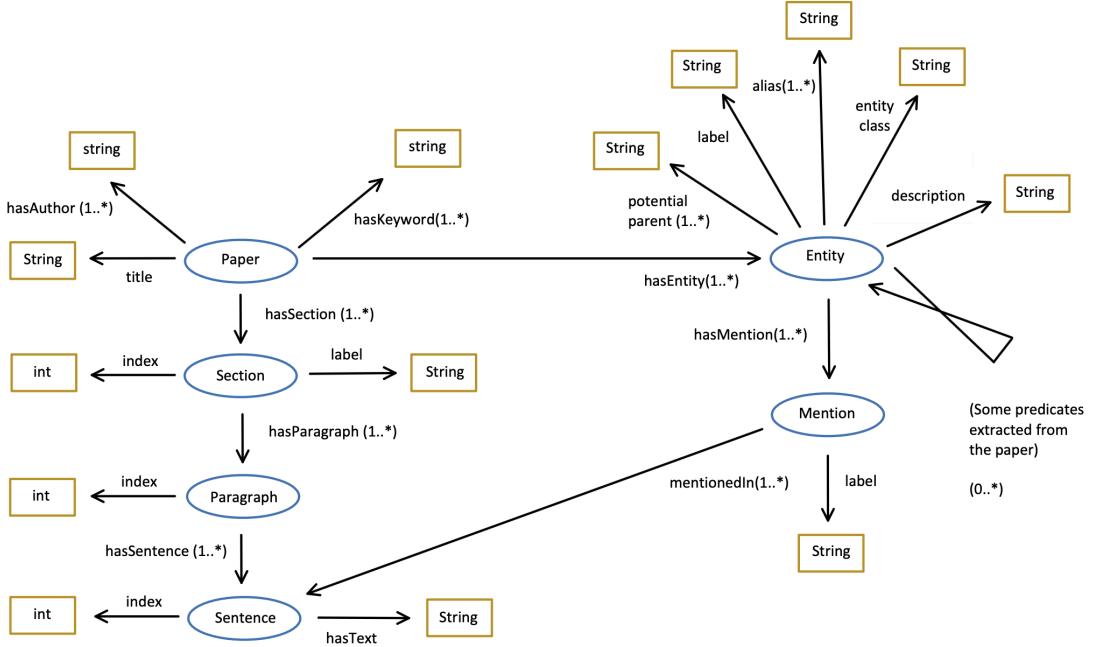


Figure 5.7: Intermediate KG Ontology after Stage 3

in academic writing, the entity that appears first is usually the full name, like “The Australian National University” instead of “ANU”, which is the most representative.

If an Other Entity is merged with a Named Entity or a General Concept, the resulting Entity will be a Named Entity or General Concept, respectively. If a General Concept is merged with a Named Entity, the resulting Entity will be a Named Entity.

After this stage, all Entity nodes in our KG are created and fixed with IRI assigned. The only things missing are the edges between Entities.

5.2.4 Stage 3: Local Relation Extraction

This stage, as shown in Figure 5.4, involves extracting the relations between entities within a limited context. For example, given that all the entities are found in a sentence, paragraph, or section, in the “(Subject, Predicate, Object)” form. we prompt the LLM decoder to extract the relations among them. During our development, we found that our lightweight LLaMA3 sometimes LLM puts predicate and object together and leaves the object slot blank. Therefore, for each triple with the object slot empty. We go through the extra triple refinement or triple decomposition step to fix these triples. Also, we check if both the subject and object of a triple exist in the document to prevent the hallucination issue of LLM, as discussed in Section 2.4.6.

The ontology of the intermediate KG after this stage is shown in Figure 5.7.

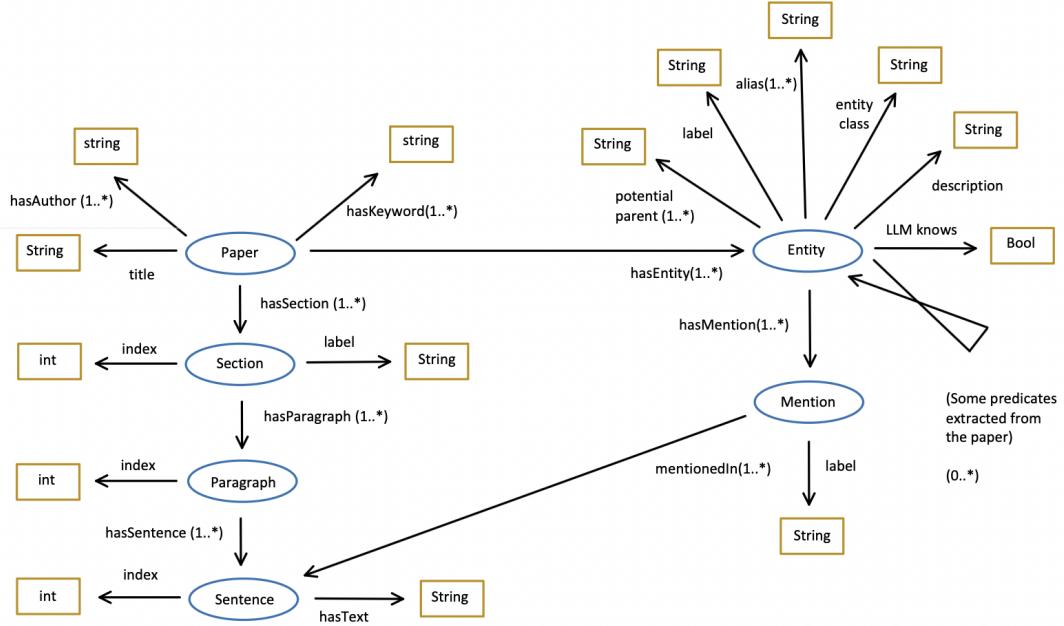


Figure 5.8: Intermediate KG Ontology after Stage 4

5.2.5 Stage 4: Global Relation Extraction

This stage, as shown in Figure 5.4, starts with shortening the original document until it fits the context limit of the LLM decoder. The shrinking is performed by prompting the LLM decoder to summarise one or more sections. The summarization is performed from the middle section of the paper up to the Abstract and Conclusion to ensure less information loss.

Then it calculates the relevancy score for each entity to the document by comparing the embedding of an entity and the embedding of a document.

Only 10% or up to 20 of the most relevant entities are selected for global relation extraction since this stage involves the complex pairwise comparison, which is $O(n^2)$. Then, for each entity pair, we fetch the LLM the shortened version of the document and extract a predicate among them.

During our development, we found that our LLM cannot effectively determine the subject and the object even if a predicate is returned. Therefore, we added another step. This step gives any two entities and their predicate extracted in the previous step to prompt the LLM decoder to determine the subject in this relation.

The ontology of the intermediate KG after this stage is shown in Figure 5.8.

5.2.6 Stage 5: Schema Generation

This stage can be divided into two parts: Taxonomy Generation and Predicate Normalisation.

Taxonomy Generation aims to create parent-child relations between entities in our graph if they have not been extracted during Stage 3 and Stage 4, such as linking “ANU” with “Public University” and “Public University” with “University”.

This step will utilise the potential parent generated in Stage 1. For all potential parents of an entity, like in Stage 2, we first generate a description and then embed it. After that, we compare these embeddings with the embeddings of the other Entity nodes in the graph. If a matching entity is found, we then say that the current entity is a child of the matching entity. This is essential for a more advanced entity typing step compared to one in Stage 1, where the types built are hierarchical instead of flattened.

Predicate normalisation is about merging similar predicates. As before, we first generate a description for each predicate and then merge them based on the similarity of their embedding. The ontology of the KG does not change after this step and maintains the one shown in Figure 5.8.

5.2.7 Stage 6: KG Post-Processing

This step is mainly about converting our customised JSON-format KG into TTL-format KG and deleting the intermediate fields, including the paper summary and potential parents. However, this is only the default option for the pipeline. Whether or not to keep these fields depends on the actual deployment. The ontology of the KG after stage 6 is the same as Figure 5.1.

5.3 Execution Diagram

The Python project structure of the pipeline is shown in Figure 5.9. It contains a master program “main.py” and a list of stage packages, where each stage package consists of a list of scripts. The execution diagram of the pipeline is shown in Figure 5.10. Since each stage has different dependencies, the master program spawns a process before running each stage to avoid too many packages loaded at once. After the process, a stage takes over control. It then imports only the packages needed during the current stage and runs each script in order. Each script first reads the result of the previous script stored on disk as “kg_x.json”. Then, it does its work and, finally, outputs another “kg_x.json” for the next script to read. In this way, all intermediate results are maintained, and if the pipeline fails, there is a recovery point. The drawback is that there is slightly more memory consumption. However, compared to tasks that involve calling LLMs, especially decoders, which can take minutes up to hours (will be shown in Section 11), the time spent on file input and output is negligible.

5.3 Execution Diagram

```
✓ LLM-KGC-DEMO
  ✓ data
    > input
    > output
  ✓ src
    ✓ stages
      > s0_kg_preprocessing
      > s1_entity_extraction_typing
      > s2_entity_resolution_disambiguation
      ✓ s3_local_relation_extraction
        > __pycache__
        > logs
        { kg_1.json
        { kg_2.json
        { kg_3.json
        { prompt_1_examples.json
        ↓ prompt_1_template.md
        { prompt_2_examples.json
        ↓ prompt_2_template.md
        { run_1_triple_extraction.py
        { run_2_triple_refinement.py
        { run_3_triple_existence_check.py
      > s4_global_relation_extraction
      > s5_schema_generation
      > s6_kg_postprocessing
      > tests
      > utilities
      { clean_output.py
      { main.py
```

Figure 5.9: Python Project Structure

5 Overview of the Implementation of the Local Holistic KGC Pipeline

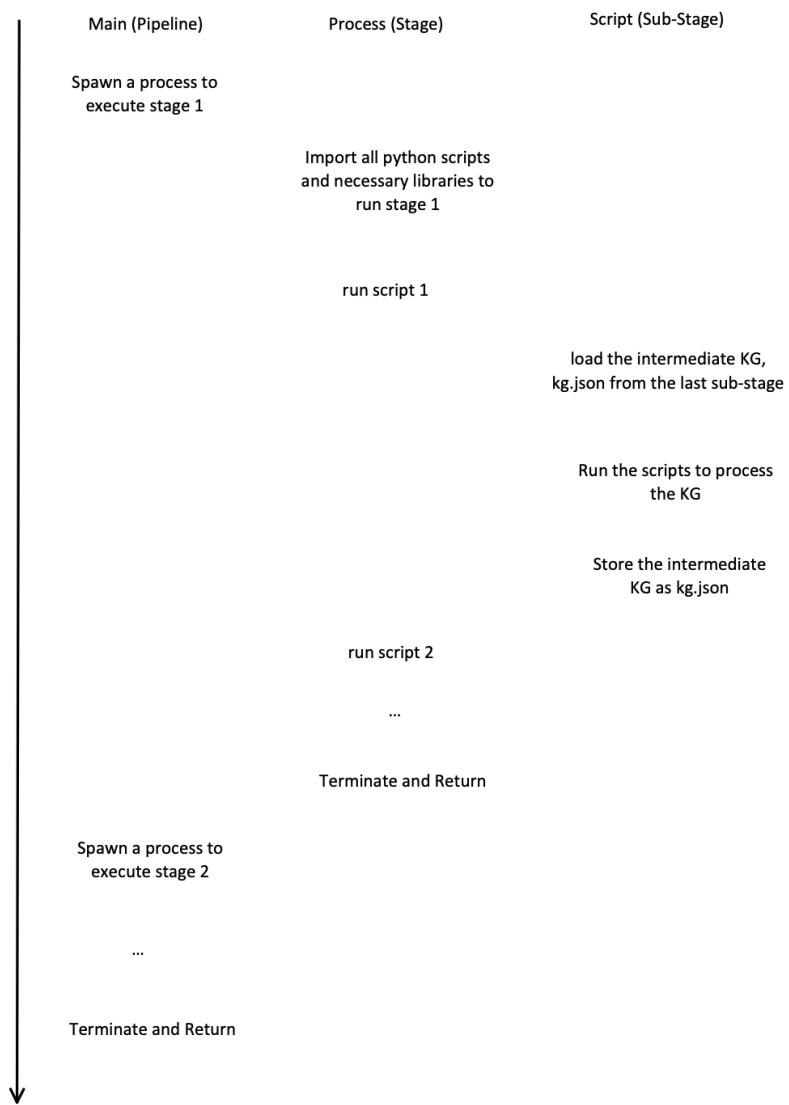


Figure 5.10: Execution Diagram of the Pipeline

5.4 Decision Made and Analysis

This section will summarise and analyse all decisions made about the structure of the pipeline.

5.4.1 Single Paper as the Primary Development Set

Pro:

- Accelerate the development process.
- Cover relatively diverse linguistic patterns compared to using only several sentences or paragraphs.
- It is a full paper. Some labelled KGC datasets do not even use a full paper, like SciERC, but only an abstract ([Luan et al., 2018](#)).

Con:

- Potentially make our pipeline biased to a single paper. Unlike traditional machine learning, no training is involved. However, the action of writing prompts by humans can be considered training. Since this paper has been read by us, our prompt can potentially be biased to this specific paper when we try to adjust our prompt to improve the performance.
- It may not be enough to discover more edge cases (i.e., the linguistic pattern that makes our pipeline fail to work as desired, e.g., cannot extract the named entity we want).
- It is unlabelled. i.e., no ground truth output KG to reference. During our research, we could not find such a labelled dataset with a mapping between a full academic paper and KG.
- Even though there are labelled datasets like SciREC and DocRED, they only consist of a limited number of predefined predicates and entity types. This does not align with our idea of open information extraction, as mentioned previously in Section [5.1.2](#).

5.4.2 Modular Design

We have kept our pipeline highly modular and stored each intermediate KG on disk instead of in memory for the following reasons:

- **Efficiency:** Different stages usually require different packages. During the execution of a stage, only the packages needed are loaded.
- **Upgradability:** Some stages can potentially be switched off (like the global relation extraction stage) and upgraded without affecting other stages.

5 Overview of the Implementation of the Local Holistic KGC Pipeline

- **Fault Tolerance:** Allow the pipelines to recover quickly in case of failure, e.g., a power outage.

The drawbacks include more disk space used, such as the code for loading and storing the intermediate files and the intermediate files themselves.

5.4.3 Using JSON instead of TTL

We use JSON instead of TTL as the format of the intermediate result of our pipeline for the following reasons:

- JSON is natively supported by Python and can be smoothly converted into Python Dictionaries in memory during the execution of the pipeline.
- Accessing the elements, e.g., an entity or sentence in Python Dictionary, only needs a simple indexing statement.
- Python Dictionary/JSON is ordered and hierarchical, which can better express the structure of a paper.
- Loading a TTL file requires the “rdflib” library to load it as an RDF-Graph Python object in memory. SPARQL queries are usually required to access elements, which can be more complex than simple indexing. SPARQL is a graph query language. It can be compared to SQL in relational databases.
- Nodes in RDF graphs have no sense of ordering. Therefore, every time we want to retrieve a list of sentences in order, we use SPARQL to sort it using SPARQL based on the index node, leading to higher complexity.
- JSON is likely more human-readable than TTL, which makes it easy for us to check the intermediate file directly without any graph visualisation tool.

The drawbacks include the extra step of converting between JSON and TTL in the first and the last stage, and our customised JSON-based format may not be easily understood by others who are already familiar with TTL.

The following compares the code for accessing Sentence 2 of Paragraph 2 of Section 2 using SPARQL (when loading TTL as RDF graph object directly) and indexing (when converting to the customised JSON-based format and loading as Python Dictionary), which demonstrates the complexity of using SPARQL for paper accessing in our scenario.

SPARQL

```
PREFIX askg-data: <https://www.anu.edu.au/data/scholarly/>
PREFIX askg-onto: <https://www.anu.edu.au/onto/scholarly#>

SELECT ?sentenceContent
WHERE {
  ?section a askg-onto:Section ;
    askg-onto:hasIndex 2 ;
    askg-onto:hasParagraph ?paragraph .
```

5.4 Decision Made and Analysis

```
?paragraph a askg-onto:Paragraph ;
    askg-onto:hasIndex 2 ;
    askg-onto:hasSentence ?sentence .

?sentence a askg-onto:Sentence ;
    askg-onto:hasIndex 2 ;
    askg-onto:hasText ?sentenceContent .
}
```

Indexing

```
paper["sections"] [2] ["paragraphs"] [2] ["sentences"] [2] ["text"]
```


Chapter 6

Stage I: Entity Mention Extraction, Classification, and Typing

The chapter will cover the implementation details of Stage 1 of the pipeline, Entity Mention Recognition, Classification, and Typing. Specifically, it will cover the algorithm, complexity and the key decisions made during the development.

6.1 Algorithm

Intuitively, as shown in Figure 6.1, Stage 1 of the pipeline operates by looping the paper document sentence by sentence, paragraph by paragraph, and section by section. In each iteration, a single sentence, paragraph, or section is sent to an LLM, and the entity mentions, along with its potential parent types, are extracted. This process is repeated three times with slightly different prompts so that the LLM can focus on “Named Entity Only”, “Named Entity plus General Concept”, and “All Potential Entity”. This not only ensures that there are minimal entity misses by focusing on different perspectives about entities but also allows the pipeline to know the specific class of entities, i.e., named entity, general concept, or other, through set subtraction. Having such entity classification may not be quite intuitive at this stage. However, this is due to some assumptions we have made in stage 3: Coreference Resolution and Entity Disambiguation (Section 7). Without having such a clear classification, our assumptions will fail. In the end, the pipeline checks whether each entity mention indeed appears in the original document, rather than imagining by the LLM. The input and output of this stage are shown in Figure 6.2, with the input components in grey and the output components in colour.

The detailed algorithm is as follows:

6 Stage I: Entity Mention Extraction, Classification, and Typing

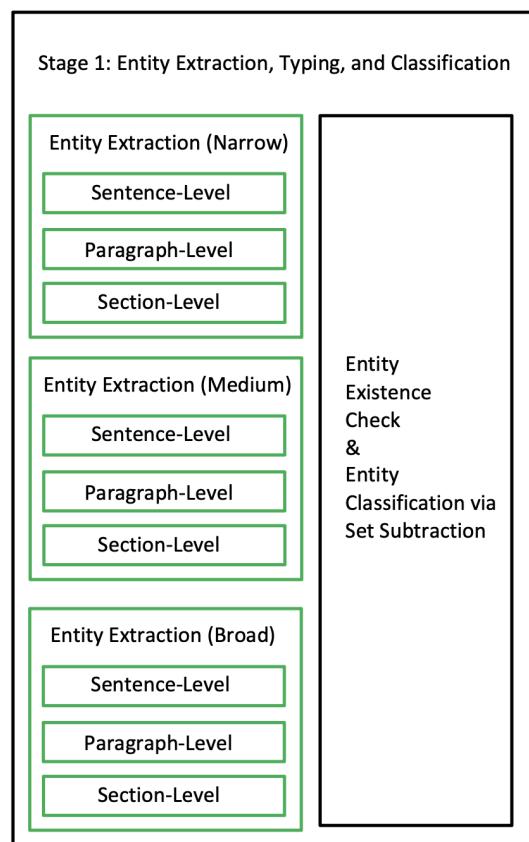


Figure 6.1: Entity Mention Extraction, Typing, and Classification (High-Level)

6.1 Algorithm

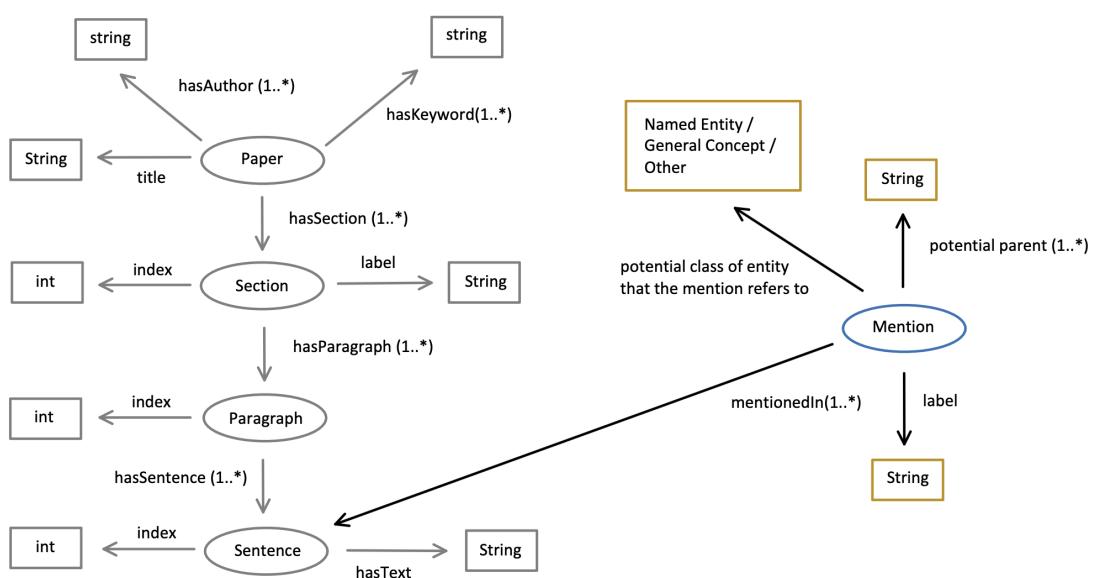


Figure 6.2: The input and output of Stage 1

6 Stage I: Entity Mention Extraction, Classification, and Typing

Algorithm

```
Input: A semi-structured paper
Output: An intermediate KG with the addition of Mentions nodes

# Extraction and Typing

For extraction_option in [Named Entity, Named Entity + General Concept,
Entity]

    For each section in the paper,

        For each paragraph in the section,

            For each sentence in the paragraph,

                * For the case of extracting Named Entities, before
                  prompting the LLM, inject a dummy Named Entity into the
                  sentence and remove it after the extraction to prevent the
                  LLM from hallucinating any Named Entities when a sentence
                  has no Named Entities

                [LLaMA Prompt_1] ask the LLM to extract all mentions
                referring to {extraction option} in the {sentence} and
                their potential parent types.

                Store them in the set called {extraction_option} directly
                under the specific {sentence} extracted from.

                [LLaMA Prompt_1] Similarly, but the {extraction_option} set is
                directly under the paragraph node.

                [LLaMA Prompt_1] Similarly, but the {extraction_option} set is
                directly under the section node.

# Clasificaton

For each section in the paper,

    For each paragraph in the section,

        For each sentence in the paragraph,

            its new "Named Entity" set = the original "Named Entity" set

            Its new "General Concept" set = the original "Named Entity +
```

6.1 Algorithm

```
General Concept" set - the new "Named Entity"

Its new "Other" set = the original "Entity" set - the new
"General Concept" set

Similarly.

Similarly.

# Existence Check and Reference Fixing

DEFAULT_NORMALISATION = lemmatisation using NTLK

def is_existing (mention, sentence):
    return whether the normalised mention is in the normalised sentence.

For each section in the paper,
    For each paragraph in the section,
        For each sentence in the paragraph,
            For each mention in {Named Entity} + {General Concept} +
{Other}:
                if not is_existing(mention, sentence),
                    discard it.

For each mention in {Named Entity} + {General Concept} + {Other}:
    For each sentence in the paragraph,
        if is_existing(mention, sentence),
            copy and add the mention to the specific set of the
            specific sentence.

Delete all mentions under the paragraph.

For each mention in {Named Entity} + {General Concept} + {Other}:
    For each paragraph in the section:
        For each sentence in the paragraph,
```

6 Stage I: Entity Mention Extraction, Classification, and Typing

```
if is_existing(mention, sentence),  
    copy and add the mention to the specific set of the  
    specific sentence.  
  
Delete all mentions under the section.  
  
# Moving all mentions from the sentence level to the paper level  
  
For each section in the paper,  
    For each paragraph in the section,  
        For each sentence in the paragraph,  
            For each mention in {Named Entity} + {General Concept} +  
            {Other}:  
                Move the mention to the paper level, i.e.,  
                paper["Mentions"]  
  
                While moving, maintain a link between the mention and the  
                original sentence from which it is extracted.  
  
                Also, maintain the information about the potential class of  
                entity that the mention refers to.
```

A key decision we made during the design of our algorithm was to inject a dummy Named Entity into the sentence during the Named Entity Mention Extraction. This was because, during development, we found that the LLaMA model we used suffered seriously from hallucination issues. When a sentence contains no Named Entities but only general concepts or other classes of entities, which is quite common (e.g., 'Cats eat fish'), the LLM then outputs anything else not belonging to Named Entities or even not in the original sentence. In other words, the LLM is “reluctant” to output an empty result. The injection is performed by appending a sentence like “This sentence is presented by “{an extremely rare person’s name}” at the end of the original sentence.

6.2 Prompts

The way we ask the LLM decoder extract mentions “Named Entity”, “Named Entity + General Concept”, and “Entity” is not about directly asking them along with these three keywords.

6.2 Prompts

Although in KGs, entities can almost refer to everything with an identity, narrowly speaking, the term “Entity” commonly refers only to “Named Entity + General Concept” in real life. Therefore, if we ask an LLM, especially the LLaMA model we used, to extract “Entities”, it will only extract “Named Entities + General Concepts”. This rule was found in multiple trial-and-error during the development. In addition, if we ask the LLM decoder to extract all “Mentions” in a prompt, it will then extract all “Entity” in our sense. Therefore, the mapping between the keywords during the extraction and the actual classes of entities that the extracted mentions refer to are as follows:

- Extracting Mentions to Named Entities -> Extracting “Named Entities”
- Extracting Mentions to Named Entities and General Concepts -> Extracting “Entities”
- Extracting Mentions to All Entities -> Extracting “Mentions”

The actual prompt template the pipeline used is as follows:

Prompt 1: Entity Extraction

Task Definition

```
You are a linguistic expert involved in a {named entity recognition/entity extraction/mention extraction} task for academic papers. Given a text in a paper, your task is to identify and extract all {named entities/entities/mentions} and assign them one or more types.
```

Output Format

```
Please write your answer in JSON format, consisting of zero or more entity-type pairs, as shown below:
```

```
"""
{format_example}
"""
```

Examples

```
**Input**: {input_example_1}
```

```
**Output**:
```

```
"""
{output_example_1}
"""
```

```
**Input**: {input_example_2}
```

```
**Output**:
```

```
"""
{output_example_2}
"""
```

6 Stage I: Entity Mention Extraction, Classification, and Typing

```
**Input**: {input_example_3}
**Output**:
```
{output_example_3}
```

## Your Task

**Input**: {text}
**Output**:

*Please write your answer*
```

Along with the instructions, we have also prepared three hand-crafted examples in order to show the LLM clearly what it is supposed to do. These examples are fixed for different sentences, paragraphs, and sections but are specific to different extraction substages, e.g., “Named Entities”, “Named Entities + General Concepts”, and “All Entities”. We set out the background at the beginning to ensure the LLM responds as a linguistic expert.

6.3 Assumptions Made

Several key assumptions were made during the development of this stage. Some of them also apply to later stages.

6.3.1 A1: LLMs are Knowledgeable

We assume that the LLM used knows all the common terminologies in NLP and linguistics. Therefore, our prompt can eliminate their explicit definitions, e.g., what a Named Entity is, in order to make it concise and reduce the complexity of an LLM call. *This assumption applies to all stages in the pipeline.*

6.3.2 A2: All Sections Fit the Context Limit of the LLM Used

We assume that the input paper has been well-preprocessed so that all its sections can fit the context limit of the LLM used by the pipeline. In addition, we also assume that every section plus the abstract of the paper (the first section) also fit within the context limit of the LLM. Having sections of appropriate length is common in academic writing, especially for published papers. If it is calculated by the pipeline that a section exists that exceeds the context limit of the LLM, the pipeline will abort, leaving its intermediate KG for recovery. *This assumption applies to all stages in the pipeline.*

6.3.3 A3: Identical Mentions in a Local Text Refer to the Same Thing

In our prompt, we ask the LLM to extract mentions in a given local text (a sentence, paragraph, or section) and then output a list of mentions found. However, we do not

6.3 Assumptions Made

track the specific location of the mentions. Therefore, we assume that all mentions with the same name within a single sentence refer to the same thing (entity). For example, consider the sentence, “An apple is the fruit of the apple tree.” two mentions of “apple” refer to the same concept. However, we acknowledge that in some rare cases, this assumption can break. For example, consider the sentence, “Apple is a technological company whose LOGO is a Bitten Apple”, if it turns out that “Apple” is extracted as a mention. there will be no way for our pipeline to figure out the exact meaning of “Apple”. Nevertheless, academic writings are unlikely to consist of ambiguous terms. If it ends up with two mentions referring to different things in the same sentence, e.g., “(Wang et al., 2024a)” and “(Wang et al., 2024b)”, we expect that the authors have disambiguated them.

The potential way to mitigate this assumption is to ask the LLM to also output the position in terms of the token index or character index of each mention, or instead of generating a list of mentions, ask the LLM to annotate the original with some symbol. The former can be hard to implement because the output of LLM is somewhat stochastic. LLM decoders operate by predating the next tokens continuing the previous tokens. Therefore, they may struggle to perform tasks requiring high accuracy (Pan et al., 2023), like counting and doing maths. Annotating the entity mentions also has disadvantages. This is because if entity mentions are nested or overlapped with each other, then annotation can become complex. Therefore, we made this assumption in our pipeline to keep things simple.

For two identical mentions in different sentences, since our data structure and the pipeline can track their locations and know their differences in locations, they do not rely on this assumption and can refer to different things.

6.3.4 A4: An Entity Mention Extracted at the Paragraph or Section Level Means All Sentence Containing the Term Same As the Entity Mention Have the Entity Mention

This assumption may not be quite intuitive. However, it basically means that if an entity mention, e.g., “ANU”, is extracted from the paragraph/section level, as long as the sentence within the paragraph/section contains the term “ANU”, the sentence will also obtain such mention, no matter mention “ANU” has been extracted from the sentence level or not.

Another example is that if the entity mention “It” is extracted from a paragraph/section, and if there are multiple “It” within the paragraph or section, we will put the mention “It” into all the sentences containing “It”. This is because having the entity “It” at the paragraph or section level is too ambiguous. We have no idea if “It” comes from sentence 1 or sentence 10. Therefore, we simply put “It” in all sentences containing “It” within the paragraph/section. By doing so, we effectively split a paragraph or section-level entity mentions into multiple sentence-level mentions. Before coreference resolution, we treat them differently and assume that they refer to different things. This is because

6 Stage I: Entity Mention Extraction, Classification, and Typing

even within a paragraph, not all identical terms, such as “It” necessarily refer to the same thing.

6.3.5 A5: All Named Entities and Concepts Extracted Are Indeed Named Entities and Concepts

We assume that when we ask the LLM to extract Named Entities, it will only return Named Entities. When we ask the LLM to extract Named Entities plus General Concepts, it will only return Named Entities plus General Concepts. This is important because our classification via the set subtraction approach relies on the correctness of the extraction process.

6.4 Limitations

There are two major limitations in this stage.

6.4.1 L1: Only Mentions That Syntactically Appear in Text Are Extracted

During the development, we found that the output of LLMs can sometimes be stochastic even with a low-temperature parameter. For example, if the original sentence is, “There are 5 people.” The output of the LLM can be “Five” instead of “5”. However, if the normalisation tool is not advanced enough, i.e., turning both “Five” and “5” into some normalised form, the mention “Five” will be rejected by our pipeline, leading to some entity miss. However, we still check if a mention extracted syntactically appears in its original text because the LLM can indeed extract something that is hallucinated by itself. Therefore, this is a trade-off.

[Wang et al. \(2023\)](#)’s study for avoiding hallucinating is that, for each entity mention extracted, prompt the LLM again to check if it is indeed a mention in the original sentence. However, not only can this lead to high complexity, but the checking result itself can also be hallucinated by the LLM, i.e., we cannot be sure that answer of “Yes” in an LLM response indeed means the entity mention exists in the original sentence.

Therefore, we tend to use the most traditional rule-based method to avoid hallucination, which leads to this limitation.

6.4.2 L2: Only Entities That Have at least One Mentions Are Captured

Our general KGC approach is to extract mentions and, from these mentions, construct entities. However, not all entities have a syntactical form in the original sentence. Like the entity “moment” in the sentence “Amy was practising piano at 5 PM”. This limitation already exists during the KGC pipeline design, as discussed in the pipeline design chapter ([4.4.2](#)), and carries through the implementation.

6.5 Complexity Analysis

This section will outline the general complexity analysis approach for all stages of the pipeline and derive the complexity of Stage 1.

6.5.1 Analysis Approach

Since an LLM decoder call can be dominantly expensive. During the modelling of our pipeline complexity, our complexity will only based on the number of times an LLM decoder is called, while other complexities, such as regular computations, file IO, or even LLM encoder calls are all modelled as $O(n)$.

Therefore, Stage 0: Input Preprocessing and Stage 6: KG Post-Processing both have a complexity of $O(n)$.

Although a single LLM decoder call is more than quadratic with respective to the number of tokens in a prompt as mentioned in the background section (2.4). Since we are loop through the document sentence by sentence, paragraph by paragraph, or section by section. The lengths of prompts are bounded and usually have similar lengths. Therefore, in our analysis, we will simply treat every LLM decoder call as $O(1)$ this will apply all the complexity analysis in the later implementation chapters.

All complexity modelled in the implementation chapter (Chapter 6 to 10) will be confirmed with the actual runtime of paper with different lengths.

6.5.2 Stage 1 Complexity

Since the algorithm of this stage loops through the paper sentence by sentence, paragraph by paragraph, and section by section, the number of times an LLM decoder is called is approximately proportional to the length of the paper L . This algorithm scans through the whole document three times in order to extract entities at three hierarchical levels. Therefore, more precisely, the complexity is $O(9L)$.

6.6 Decisions Made and Analysis

The following subsections are about the key decision made and their reason.

6.6.1 Simple Prompts vs. Detailed Prompts

We tend to write prompts as simply and concisely as possible. This applies to all stages of the pipeline. This is because LLM complexity is more than quadratically proportional to number of tokens in the prompt. During our development, we indeed created both a short version and a long version for the prompts. However, it turned out that an LLM could respond to a short prompt within a second. In contrast, a long prompt can take up to half a minute, which significantly affects the runtime performance since our pipeline makes plenty of prompts to the LLM during its execution.

6 Stage I: Entity Mention Extraction, Classification, and Typing

In addition, different people may have slightly different understandings of some terminologies, such as named entities. This also applies to humans and AIs. If we prepare a long prompt trying to define what a named entity is, but such a definition contradicts the internal knowledge of the LLM, the LLM may get confused. Therefore, we leave the definition of terminologies to the LLM and trust the LLM. However, the drawback is that we have to assume LLMs are knowledgeable.

The decision above applies to all the later stages.

Although [Xie et al. \(2023\)](#)' study suggests that we can add part-of-speech tagging along with the original text to assist the LLM in extracting the named entity based on both semantic and syntactical reasoning. However, during our development, we have found that adding Part-of-Speed only leads to minimal impacts. However, since the prompt became longer, the runtime significantly increased. Therefore, we decided to keep our prompt simple and expect the LLM to do the task without the assistance of Part-of-Speech tagging.

[Wang et al. \(2023\)](#)'s study suggests that the supplied examples in the prompt should be relevant to the input text. For example, if we want to extract the named entities from the sentence, "ANU is a university located in Canberra." the supplied examples in the prompt should be drawn from some knowledge base that is semantically relevant to the input sentence. However, most currently existing datasets that can serve as a knowledge base for entity extraction only consider entities that fall within a predefined set of types. Therefore, these examples can be misleading because our goal is to let LLM extract anything from the sentence as long as it is an entity without restricting it to some predefined types. Therefore, the use of a knowledge base to retrieve relevant examples is not practical during open entity extraction. Instead, we decided to provide only three general and fixed hand-created examples. The main goal is to only let the LLM be familiar with the task definition instead of teaching precisely what is supposed to be extracted as an entity and what is not. This is also another decision regarding making prompts as simple as possible.

6.6.2 Hierarchical Extraction

Our pipeline performs entity extraction hierarchically in two dimensions: entity and text.

Extracting entities through all sentence, paragraph, and section levels allows the LLM to have both a narrow view and a broader view of the text so that there are fewer entities missed.

Figures [6.3](#), [6.4](#), and [6.5](#) show the comparison between sentence-level extract extraction and paragraph-level entity extraction for the same part of the text of the paper in our development set. Figures [6.3](#) and [6.4](#) consist of the sentences from the paragraph contained in Figure [6.5](#). We would simply refer to them as sentence 1, sentence 2, and paragraph 1, respectively.

6.6 Decisions Made and Analysis

Your Task

Input: This manuscript introduces J2RM: a tool to process mappings from JSON data to RDF triples guided by an OWL2 ontology structure.

Output:

Please write your answer



GPT4ALL Llama 3 8B Instruct

Here is the output in JSON format:

```
[  
  {  
    "entity": "J2RM",  
    "types": [  
      "tool",  
      "software"  
    ]  
  },  
  {  
    "entity": "JSON data",  
    "types": [  
      "data",  
      "format"  
    ]  
  },  
  {  
    "entity": "RDF triples",  
    "types": [  
      "data",  
      "format"  
    ]  
  },  
  {  
    "entity": "OWL2 ontology structure",  
    "types": [  
      "ontology",  
      "knowledge representation"  
    ]  
  }  
]
```

Let me know if you have any questions or need further clarification!

Figure 6.3: Sentence-Level Entity Extraction

Note that the prompt is only partially captured in the screenshot due to the size, but they are similar to the prompt template shown previously. The tool we used is the GUI version of GPT4ALL. Although our pipeline uses the Python package version of GPT4ALL, They function similarly.

For sentence 1 shown in Figure 6.3, it was found that most entities are extracted except “manuscript”. However, “manuscript” was captured by the paragraph-level extraction in Figure 6.5.

In addition, many entities are missed during the paragraph-level extraction, such as “RDF-Graph creation”. However, the mention “RDF-Graph creation” was successfully captured by the sentence-level extraction as shown in Figure 6.4.

Therefore, the different hierarchical levels of entity extractions can complement each other, similar to the hierarchical entity extraction from narrow entity class to board entity class. However, the key reason for the hierarchical entity extraction of different broadness is to perform entity classification via set subtraction.

6 Stage I: Entity Mention Extraction, Classification, and Typing

Your Task

Input: They are embedded in an ontology file so that they can be readily deployed and shared to automate RDF-graph creation. **Output:**

Please write your answer



GPT4All Llama 3 8B Instruct

Here is the output in JSON format:

```
[  
  {  
    "entity": "ontology file",  
    "types": [  
      "file",  
      "data source"  
    ]  
  },  
  {  
    "entity": "RDF-graph creation",  
    "types": [  
      "process",  
      "algorithm"  
    ]  
  }  
]
```

Figure 6.4: Sentence-Level Entity Extraction 2

6.6.3 Explicit Entity Classification vs. Classification via Set Subtraction

During the development, we found that directly asking the LLM to classify entities into “Named Entity”, “General Concept”, and “Other” does not work well. Unlike LLM encoders, LLM decoders struggle to perform classification tasks. This is mentioned in [Edwards and Camacho-Collados \(2024\)](#) and [Sun et al. \(2023\)](#)’s study. This is why we choose to perform the implicit classification via set subtraction.

We have tried two explicit entity classifications. One is the classified entities when they are extracted. That is, in the prompt, we explicitly ask the LLM to extract either Named Entities, General Concepts and Other that do not belong to Named Entity and General Concept. However, this approach does not work because we found that the more strict and specific the instruction, the worse the performance the LLM has, which is likely due to the hallucination issue. As mentioned earlier. Extracting named entities only is already difficult for the LLM, as we have to inject a dummy named entity to ensure the LLM does not hallucinate. Therefore, extracting general concepts or other entities only can also be hard.

The second way to perform explicit entity classification is to first prompt the LLM to extract all entities regardless of their class. Then, we make another prompt to ask the LLM what the entity belongs to, either named entities, general concepts, or others. It ended up that the LLM could easily make wrong predictions.

Therefore, our final solution is to gradually increase the scope of the entity required for the LLM to extract. In this way, not only do we perform a basic entity classification via

Your Task

Input: This manuscript introduces J2RM: a tool to process mappings from JSON data to RDF triples guided by an OWL2 ontology structure. The mappings are defined as annotation properties associated with each ontology entity of interest. They are embedded in an ontology file so that they can be readily deployed and shared to automate RDF-graph creation. In this paper, we motivate the need for such mappings, describe some of their definitions on a use case example, present the formal grammar of the mapping language, and explain how these mappings work. We conclude with a discussion of the key aspects, main contributions, and future improvements. **Output:**

Please write your answer



GPT4All Llama 3 8B Instruct

Here is the output in JSON format:

```
[
  {
    "entity": "J2RM",
    "types": [
      "tool",
      "software"
    ]
  },
  {
    "entity": "JSON data",
    "types": [
      "data",
      "format"
    ]
  },
  {
    "entity": "RDF triples",
    "types": [
      "data structure",
      "semantic web"
    ]
  },
  {
    "entity": "OWL2 ontology structure",
    "types": [
      "ontology",
      "knowledge representation"
    ]
  },
  {
    "entity": "annotation properties",
    "types": [
      "metadata",
      "data annotation"
    ]
  },
  {
    "entity": "ontology entity of interest",
    "types": [
      "concept",
      "knowledge concept"
    ]
  },
  {
    "entity": "manuscript",
    "types": [
      "academic paper",
      "research document"
    ]
  }
]
```

Figure 6.5: Paragraph-Level Entity Extraction

6 Stage I: Entity Mention Extraction, Classification, and Typing

set subtraction, but we also mitigate the shortage of classification in LLM decoders.

6.6.4 JSON Output vs. Customised-Format Output

Last but not least, because the response of LLM decoders is plain text, we have to parse the response back to structured data type, such as a list of entity-type pairs, in order for the algorithmic part of the pipeline to process.

Originally, we defined our customised format for the output in the prompt, i.e., using “[” and “]” or some special to mark a list so that our parser can successfully parse the entity lists in the LLM’s response. However, it concluded that there always exists some edge case that prevents successful parsing. Later, we found that LLM decoders seem to be good at providing JSON output, and the parsing tool from JSON into Python Dictionary has already been built into it. Many edge cases can also be automatically handled by JSON through escaping. Therefore, we decided to make the response of all LLM calls in JSON format.

In addition, we decided to use the Markdown syntax to write all the prompts in the pipeline because we found that the LLM decoder used, i.e., LLaMA, can properly understand and handle Markdown. There are also studies showing that a more structured syntax can help an LLM understand the prompt better compared to plain text ([Wang et al., 2024](#)).

Chapter 7

Stage II: Entity Disambiguation and Coreference Resolution

This chapter will cover the implementation details of Stage 2: Entity Disambiguation and Coreference Resolution by discussing the algorithm, complexity, and key decisions made.

7.1 Algorithm

Intuitively, as shown in Figure 7.1, the algorithm starts by creating an entity node in the intermediate KG for all mentions. That is, it conservatively treats all mentions as different things, e.g., “Python” in sentence 1 is different from “Python” in sentence 14. Therefore, every entity only has one mention, for now, where the entity’s label (name) is the same as the mention’s label.

Then, Entity Disambiguation is performed. This step is about carefully merging two entities with the same name. It is mainly based on the assumption that “Named entities

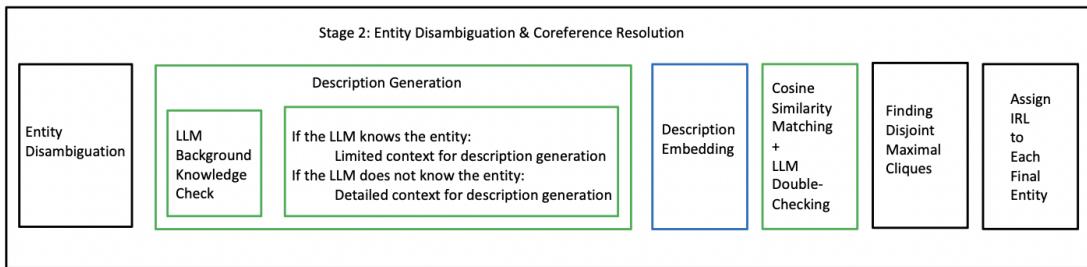


Figure 7.1: Entity Disambiguation and Coreference Resolution (High-Level)

7 Stage II: Entity Disambiguation and Coreference Resolution

and general concepts in academic writing are unambiguous”. For example, we assume that all occurrences of “Python” in academic writing refer to either a type of snake or a programming language. All occurrences of “Wang” refer to the same person. The general concept “Plane” refers to either a flat surface or a flying vehicle, but not both. We expect that the author of the paper has appropriately handled any ambiguity, e.g., adding brackets for any potentially ambiguous term. Therefore, with this assumption, we can safely merge all entities belonging to named entities and general concepts with the same name. When merging two entities, their mentions set are combined. The entity label will be the label of the mention that appears first in the paper. This is also based on the assumption that academic writing tends to only provide the full name of a term, especially a terminology, at the beginning and use its acronym throughout the rest of the text. However, the full name should be the most representative name of an entity. Other names that are not used as labels then become aliases.

Even though we merge named entities and general concepts, we keep all Other entities untouched because our assumption does not hold for all types of entities. For example, consider an entity “It” created from a mention in the original paper. Without Coreference Resolution, we cannot be sure if it is a named entity, general concept or neither. An “It” in sentence 1 is clearly not necessarily the same as the entity “It” in sentence 14. If a named entity is merged with a concept, we treat it as a named entity.

After merging entities with the same name, we then try to merge entities with different names but refer to the same thing, i.e., coreference resolution. Like what LLM-based coreference resolution is commonly performed, as discussed in the literature review section (3) our coreference resolution is based on comparing the embeddings of any two entities. If they are highly similar and the result is double-checked by the LLM decoder, we then draw a temporary link between them. However, this does not mean they must be merged into the same entity. This is because if A is similar to B and B is similar to C, it does not mean A is similar to C since the nuances between two entities cannot be easily detectable, and there exists a threshold between treating them as two different things or the same thing. Our algorithm groups multiple entities into the same entities only if their temporary links form a complete subgraph, and we also hope the groups are as large as possible, i.e. finding maximal cliques. However, we also hope each clique is disjointed. Therefore, we iteratively call the Bron-Kerbosch algorithm ([Bron and Kerbosch, 1973](#)) to find the maximum cliques and remove all the nodes belonging to the clique in the main graph. After merging, we assign a unique IRI for each entity and mention it so that our intermediate KG can eventually become a valid RDF-KG.

To obtain embedding for each entity, we use the entity labels, aliases, potential parent types, and most importantly, its description. A good description should clearly define what an entity is and be able to distinguish it from other entities with the same name. It should also provide essential background knowledge about the entity if the entity is newly introduced by the paper and unseen by the LLM.

To generate a high-quality entity, we first perform a background check for the LLM

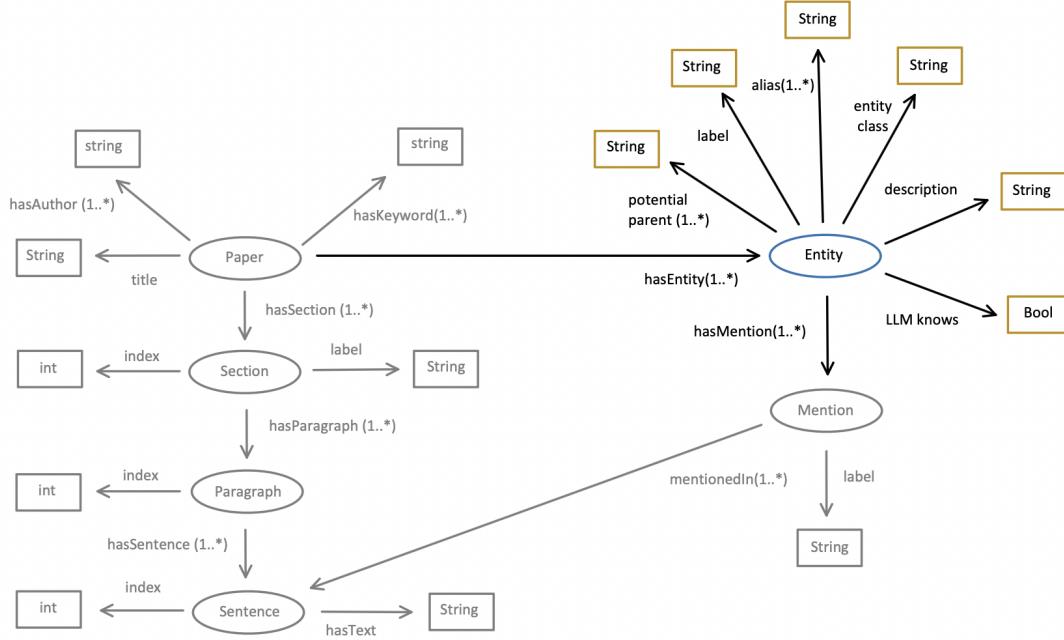


Figure 7.2: Input and output of Stage 2

to see, for each entity, if the LLM knows it without the assistance of a context. This step can effectively mark all new entities, e.g., any new terminologies introduced by the paper and unseen by the LLM, as well as any potentially ambiguous entity, such as “It”. When generating a description, if the LLM knows the entity, limited context will be given to the LLM in order for the description to be as general as possible, such as the entity “University”. If the LLM does not know the entity, detailed context will be given, including the title, authors, and abstract of the paper, as well as the specific section where the entity is located, such as the entity “J2RM” introduced in the paper in our development set. For Other entities, no matter if the LLM is familiar with it, our pipeline always supplies it with detailed context. In this case, with sufficient recourse, the description for any new or ambiguous entity, like “J2RM” can achieve a high quality. When we embed an entity along with its description, the embedding is then also representative.

The input and output of this stage are shown in Figure 7.2, with the input components in grey and the output components in colour.

7 Stage II: Entity Disambiguation and Coreference Resolution

The detailed algorithm is as follows:

Algorithm

```
Input: An intermediate KG with Mention nodes
Output: An intermediate KG with the addition of Entity nodes

# Entity Creation

For each mention in the KG,
    create an entity node, where its label is the same as the label of the
    mention, its alias set contains only its label, its potential parent
    set is the same as the mention's one, and its entity class is the same
    as the potential entity class of the mention.

    Create a link between the newly created entity node and the original
    mention node.

    Delete all attributes of the original mention node except its label,
    serving as the local name of the entity in a specific sentence.

    Create a description attribute for the entity as left empty.

# Entity Merging Function

def merge_entity(entity_1, entity_2):
    new_potential_parents = union of the entity_1's parents and entity_2's
    new_mentions = union of the entity_1's mentions and entity_2's
    new_aliases = union of the entity_1's aliases and entity_2's

    new_label = the label of the Mention in new_mentions that is mentioned
    first in the paper.

    new_description = the description of the Entity with the Mention in its
    original_mention set that is mentioned first in the paper.

    new_entity_class = the original entity classes of the two entities with
    the high precedence. (named entity > general concept > other)

    return construct a new entity with the new attributes and delete the
    two old entities.

# Entity Disambiguation

For entity_1 in the KG,
    for entity_2 in the KG,
```

7.1 Algorithm

```
if entity_1 != other and entity_2 != other,  
    if entity_1["label"] == entity_2["label"],  
        draw a temporary link between them.  
  
Merge all entities connected by temporary links using merge_entity.  
  
# LLM Background Knowledge Check  
  
For each entity,  
    [LLaMA Prompt_1] Given the entity's label only, ask the LLM if it can  
    confidently know the exact entity the entity label refers to.  
  
# Description Generation  
  
For each entity in the KG,  
    if entity["LLM_knows"] == False or entity["class"] = "Other",  
        [LLaMA Prompt_2.1] Use the entity's label, potential parent types,  
        the paper's title, the paper's authors, the paper's keywords, the  
        paper abstract (first section), the section-level context (the  
        section where the entity appears), the sentence-level context (the  
        specific sentence in the section where the entity appears) to  
        generate a short description.  
    else,  
        [LLaMA Prompt_2.2] Use the entity's label, potential parent types,  
        sentence-level context to generate a short description.  
  
# Embedding Generation  
  
For each entity in the KG,  
    [BGE] use its label, potential parent types, and description to  
    generate an embedding.  
  
# Coreference Resolution  
  
For entity_1 in KG["Entities"],
```

7 Stage II: Entity Disambiguation and Coreference Resolution

```
for entity_2 in KG["Entities"] ,  
    if entity_1_id < entity_2_id,  
        compute the cosine similarity between the embeddings of  
        entity_1 and entity_2.  
  
    If similarity > 0.9,  
  
        [LLaMA Prompt_3] Given the paper's title, authors, and  
        keywords, given the labels, potential parent types,  
        description, section-level context, and sentence-level  
        context of both entities, ask the LLM to determine if they  
        are the same.  
  
        * If having two section-level contexts is too long for the  
        context limit, use paragraph-level contexts instead.  
  
    If the LLM says yes,  
  
        Create a temporary link between the two entities.  
  
# Entity Merging  
  
disjoint_maximal_cliques = []  
  
In the graph of related entities, use the Bron-Kerbosch algorithm to find  
a set of (non-disjoint) maximal cliques.  
  
Select the largest clique within the set of maximal cliques.  
  
Remove all the nodes in the original related entity graph that are in the  
selected maximal clique.  
  
Add the selected maximal clique into disjoint_maximal_cliques.  
  
Repeat until no more maximal cliques can be found with sizes larger than 1.  
  
For each clique in disjoint_maximal_cliques,  
    merge all entities using merge_entity.  
  
# Assign IRIs  
  
For each entity,
```

assign it an IRI based on its label for readability but ensure uniqueness.

assign an IRI for all its mentions based on the IRI of the entity.

7.2 Prompts

The following are the simplified versions of the three LLM prompts mentioned in the algorithm.

Prompt 1: LLM Knowledge Check

Task

Assume that you are reading an academic paper. Given a term extracted from the paper and its types, without seeing the context, can you tell me what the term "{term}" refers to based only on your current knowledge?

- Answer true if you are confidently sure what it is.
- Answer false if you do not know the term, or it is potentially ambiguous, i.e., it may have several meanings, but its specific meaning cannot be determined without seeing the context.

Output Format ...
Term ...
Answer ...

Prompt 2.1: Description Generation (Detailed)

Task

You are a linguistic expert involved in a description writing task for an entity mentioned in a paper. Given an entity's mention "{label}", its types, the abstract of the paper (paper-level context), the specific section where the mention appears (section-level context), and the specific sentence of the entity mention (sentence-level context), your task is to write a single-sentence description that can precisely tell what the actual entity the mention "{label}" refers to.

Output Format ...
Input ...
Output ...

Prompt 2.2: Description Generation (General)

Task

You are a linguistic expert who is involved in a description writing task for an entity mentioned in a paper. Given an entity's mention "{label}", its types, and the specific sentence of the entity mention (context), your task is to write a single-sentence description that can precisely tell what the actual entity the mention "{label}" refers to.

- Your description should be as general as possible and not be specific to the given context.
- The context is mainly used to help you understand the entity or disambiguate the entity with others with the same name.

Output Format ...

Input ...

Output ...

Prompt 3: LLM Double Check

Task

You are a linguistic expert who is involved in the task of disambiguating two terms in a paper. Given the abstract of the paper (paper-level context), the term "{label_1}" and the term "{label_2}", each with its label, types, description, and the specific section where the mention appears (section-level context), and the specific sentence of the entity mention (sentence-level context), your task is to determine whether these two terms refer to **exactly** the same entity or concept. Please provide a short explanation and then answer true or false in JSON format, e.g., ...

Input ...

Output ...

7.3 Assumptions Made

There are a number of assumptions made in this stage:

- When merging entities by names, we assume that all named entities and general concepts in the paper are unambiguous, i.e., named entities with the same label always refer to the same thing, which is highly likely in academic writing.
- When deciding the label of an entity, we assume that the mention of an entity that appears first in the paper always carries the most representative name for the entity.
- When using the LLM to generate a description for an entity that the LLM does

not know, we assume that giving the paper’s title, authors, keywords and the entity’s label, types, section-level context, and sentence-level context is enough for the LLM to understand the entity so that the description generated is accurate.

- When merging two entities and deciding the description of the new entity, we assume that the description of the original entity with its mention that first appears in the paper is the most representative description.
- When merging two entities and there is a conflict between the entity class of the new entity, we assume that classifying as Named Entity is always more accurate than classifying as General Concept, which is more accurate than classifying as Other Entity.

7.4 Limitations

Several limitations can be found during this stage.

- We used embeddings to check if two entities are the same. However, the embeddings themselves can be lossy. That is, two close embeddings may not always represent two close entities in the real world.
- We used description to generate embeddings for entities. However, the descriptions generated by the LLM decoder are already lossy.
- When checking the LLM familiarity of an entity, the LLM may not answer honestly if it knows the entity or not. It can be too optimistic or conservative. This is because the behaviours of LLMs are stochastic, and they may suffer from the hallucination issue ([Pan et al., 2023](#)).
- When asking the LLM if two entities are the same during the double-checking process, the LLM may not answer honestly if two entities are the same. It can be too optimistic or conservative.

7.5 Complexity Analysis

The algorithm involves looping through all entities (not fully merged) twice to check if the LLM are familiar with them and generates a description. Therefore, the complexity here is $O(2M)$, where M is the number of mentions. This the process of entity merging has yet finished before this stage. We choose to use the number of mentions instead of the number of entity. Even though we can create an entity for each mention at the beginning of this stage, the number of entities is not yet final. That is why we use M instead of E .

Although the process of generating embedding for entities involves LLM encoders, since LLM encoders are generally fast and can be done in a second during the development,

7 Stage II: Entity Disambiguation and Coreference Resolution

we simply ignore their complexity. Similarity for the pairwise comparison for their embeddings.

However, although we only ask the LLM decoder to double-check only when two entities are found to be highly similar, in the worst-case scenario, it is still $O(E^2)$. Therefore, the overall complexity of this stage is $O(2E + E^2)$.

7.6 Decisions Made and Analysis

Two major decisions were made during the development of this stage. One is about the use of description in generating embedding. The other is about using disjoint maximal cliques instead of connected subgraphs to determine if a group of entities should be merged.

7.6.1 Using Description for Embedding Generation

During the implementation of the pipeline, we found that the description of an entity is the most valuable resource for Entity Disambiguation and Coreference Resolution. This is because even though two entities have the same name, like “Apple” (a company) and “Apple” (a fruit). Their description should be clearly different. The second most valuable resource is types. Clearly, the two “Apples” in the example have two different types. Our findings are further confirmed by some literature about entity disambiguation and coreference resolution, such as [Salman et al. \(2024\)](#), [Li et al. \(2020\)](#), and [Ayoola et al. \(2022\)](#)’ studies, where entity description or entity types are used when distinguishing two entities.

Traditionally, when Entity Disambiguation and Coreference Resolution are performed solely based on LLM encoders, the context may also be used directly to generate embedding ([Joshi et al., 2020](#)). However, this requires some fine-tuning for the LLM encoders because the general-purpose encoder we used, i.e., BGE and M3, can only embed the input as a whole. Therefore, if the context is used, the LLM may be distracted, i.e., focusing on generating the embedding for the context instead of the entity itself. Since our research tends to do everything purely based on in-context learning without fine-tuning any models, we choose to use description for embedding generating. In this case, everything input to the general-purpose LLM encoder is highly relevant to the entity. Therefore, the embedding generated should also accurately represent the entity.

The generated description is helpful not only in Coreference Resolution and Entity Disambiguation but also in the later stages, such as relation extraction. Essentially, whenever we supply the LLM with an entity, we always attach its description and sometimes its types. This is because only the entity label (name) is insufficient for the LLM or even humans to determine what it is. The description can also help the LLM to understand the LLM even though the entity is new to the LLM.

7.6.2 Disjoint Maximal Cliques vs. Disjoint Connected Subgraphs

In our initial implementation, we merge any two entities as long as they are related, i.e., based on connected subgraphs. However, the result is that many clearly distinct entities are merged together, such as “RDF-Graph” and “J2RM” (a tool to convert JSON into RDF). This is because there are often nuances between two entities with different names. However, if A and B are close enough to become the same entity and B and C are also close enough, it does not necessarily mean A and C are close. The worst case is that multiple entities form a hyper-line in the vector space of their embeddings, where the entities at either end are clearly different. Therefore, we merged a group of entities only when they formed a clique in the related entities graph. This is similar to clustering. However, we strictly control the diameter of each cluster.

Chapter 8

Stage III: Local Relation Extraction

This chapter will present the implementation details of Stage 3: Local Relation Extraction in a similar way to the previous chapters, i.e., the algorithms, prompts, assumptions made, limitations, complexity, and decisions made.

8.1 Algorithm

The way how local relation extraction works at a high level is similar to entity extraction. As shown in Figure 8.1, the algorithm starts by looping through the paper sentence by sentence, paragraph by paragraph, and section by section. Within each iteration, a text (sentence/paragraph/section) along with all the entities that have mentions linking to the text is input to the LLM. The output of the LLM is a list of triples, where the subject and the object come from the given entities and the predicate is extracted on the fly. During the implementation, we found that the LLM model we used, i.e., LLaMA, can sometimes return only the subject and the predicate, where the predicate consists of both the verb and the object, leaving the actual object slot of the triple empty. This is potentially because the term “predicate” is ambiguous and may refer to the “verb” only or the combination of “verb” and “object”. However, even if we explicitly ask the LLM to extract (subject, verb, object) triples, the problem persists. Therefore, we add another layer, called triple refinement, which prompts the LLM again to further decompose a triple with its object slot empty. The algorithm ends with the entity existence check, ensuring that the subject and object of all triples can all map to the entity nodes we created in the last stage.

The input and output of this stage are shown in Figure 8.2, with the input components in grey and the output components in colour.

The algorithm is as follows:

8 Stage III: Local Relation Extraction

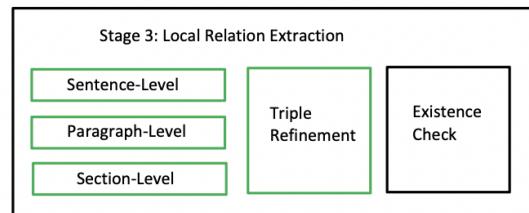


Figure 8.1: Stage 3: Local Relation Extraction

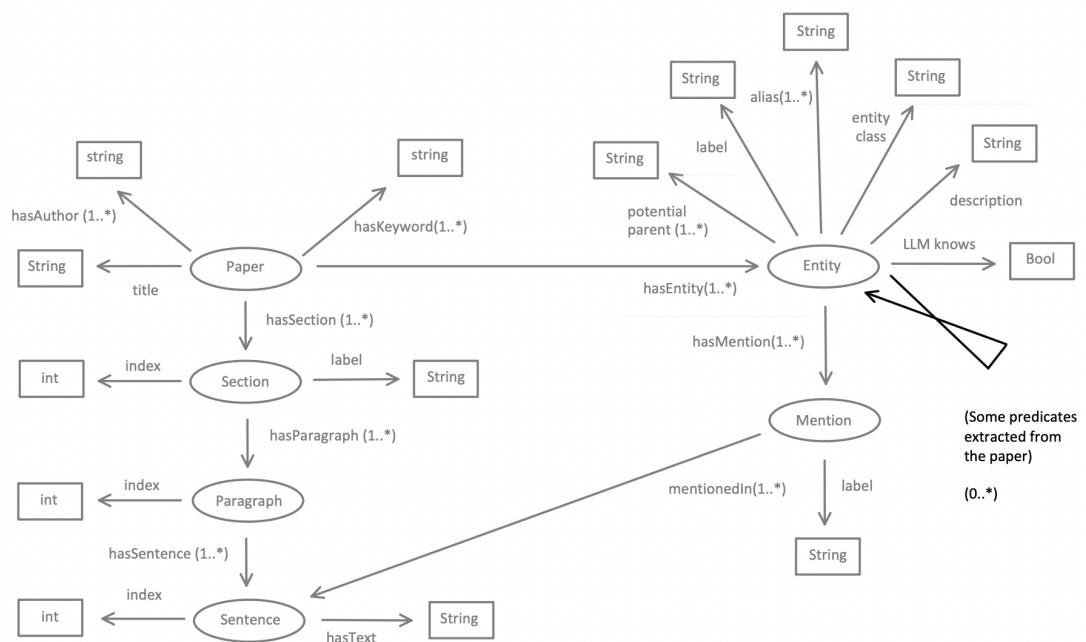


Figure 8.2: Input and Output

Algorithm

Input: An intermediate KG with entity nodes created.
 Output: An intermediate KG with the addition of some predicates connecting between its entity nodes.

Creating inverse references from sentence/paragraph/section to entity.

For each entity in the KG,

for each mention of the entity,

find the sentence where the mentions refer to.

Put the entity IRI and the entity's local name (the mention's label) into a set under the sentence, as well as the set under the paragraph and section where the entity is mentioned.

Local Relation Extraction

For each section in the paper,

for each paragraph in the section,

for each sentence in the paragraph,

given a set of (entity_IRI, local_name) pairs, try to create an inverse mapping between the local names and entity IRI.

If there exist multiple identical local names referring to different entity IRI,

Modify the local to make it unique by adding some numbers, e.g., (1)

[LLaMA Prompt_1] Given the sentence and all the local names of all entities that the sentence contains, ask the LLM to extract the relations among them. Return the result in the form of a list of triples, where the subject and object must be chosen from the given list of local names of entities.

Store the triple under the sentence temporarily.

[LLaMA Prompt_1] Similarly.

[LLaMA Prompt_1] Similarly.

8 Stage III: Local Relation Extraction

```
# Triple Refinement

For each section in the paper,
    for each paragraph in the section,
        for each sentence in the paragraph,
            for each triple in the sentence,
                if its object is empty,
                    [LLaMA Prompt_2] Given the problematic triple, ask the
                    LLM to further decompose it into (Subject, Verb,
                    Predicate).

# Triple Existence Check

For each section in the paper,
    for each paragraph in the section,
        for each sentence in the paragraph,
            by using the mapping created before, find the actual entity
            IRI of the subject and the object of the triple.

            # This turn a triple of (local_name, predicate, local_name)
            into a triple of (IRI, predicate, IRI)

            If either the subject name or the subject name has no
            corresponding IRI,
                discard the triple.
            Else,
                move the triple from the sentence level to the paper level.

            Similarly.

            Similarly.

            Remove any redundant triples (triples with the same subject, predicate,
            and object) at the paper level.
```

8.2 Prompts

The following are the simplified versions of the three LLM prompts mentioned in the algorithm.

Prompt 1: Local Relation Extraction

```
## Task Definition
```

You are a linguistic expert involved in a relation extraction task for an academic paper. Given a text in the paper and a list of terms appearing in the text, your task is to extract the relations between these **terms** based on the context.

```
## Output Format ...
```

```
## Examples ...
```

```
## Your Task ...
```

Prompt 2: Entity Refinement

```
## Task Definition
```

You are a linguistic expert whose job is to decompose (S,VO) pairs into (S,V,O) triples, i.e. (Subject, Verb, Object) triples. Write your answer in JSON format.

```
## Examples ...
```

```
## Your Task ...
```

8.3 Assumptions Made

We assume that the relations, in the form of triples, returned by the LLM are indeed the relations that exist in the paper rather than something hallucinated by the LLM. We made this assumption because, unlike checking the existence of the entity, which can simply be performed via string matching or fuzzy matching, it is difficult to check the existence of a relation as well as the predicate in the relation. This is because we allow the predicate returned by the LLM to not necessarily syntactically appear in the document in order for the LLM to perform relation extraction freely to extract more relations. For example, the original text does not need to explicitly say that “ANU is located in Canberra” for the relation “(ANU, located in, Canberra)” to be extracted. This is because there are many ways to implicitly and semantically express such relations in a text. If we simply ensure all predicates must be in the original text, we will lose plenty of relations.

Due to this openness, there is also a possibility that the relation is hallucinated by the LLM. However, we assume that it is rare.

8.4 Limitations

Although we do not perform the existence check about the predicate of a relation, our algorithm still makes sure that the subject and object of a triple should at least be from the entities extracted before.

However, due to the stochastic behaviour of LLMs, the subject and object of triple returned sometimes do not have any matching entities. The subject and object can be the following three cases:

- It does not exist in the original text, neither syntactically nor semantically.
- It does not syntactically exist in the original text.
- It exists syntactically in the original, but it is missed by Stage 2: Entity Extraction of the pipeline.

To solve case three, we may add it to the graph as a new entity, and we may perform Stage 2: Coreference Resolution and Entity Disambiguation again. However, since there are multiple cases in which an extracted triple has a subject or object that does not match any existing entity in the graph, our pipeline chooses to discard the triple for simplicity, which leads to this limitation.

8.5 Complexity Analysis

The pipeline involves scanning through the paper 3 times at different hierachal levels. Therefore, the complexity involves $O(3L)$. After the initial extraction, we call the LLM again for any triple with an empty object. This is the upper bound by the number of triples extracted in the worst case. Therefore it is $O(R_1)$, where R_1 is the total number of relation (triples) so far. Since the number of relations can still grow in Stage 4: Location Relation Extraction, we use R_1 to only refer to the relations extracted in Stage 3, the current stage.

8.6 Decisions Made and Analysis

The major decision made was to use the local names of the entities in the prompt of asking the LLM to extract relations between the entities within a text. Using local names ensures that the entities provided in the prompt align with the provided text, which prevents the LLM from confusing. An entity can have different forms, and not all its forms appear in the text from which we want to extract the relations. The challenge is that the same local name may map to different entities. Therefore, disambiguation is required so that when an LLM outputs a triple consisting of two local names and one predicate, we can know the actual entities the local names refer to.

The second decision is about hierarchical extraction. This reason is similar to hierarchical extraction for entities, which allows the LLM to both have a specific and broad view of

8.6 Decisions Made and Analysis

a text, potentially making more relations extracted.

Chapter 9

Stage IV: Global Relation Extraction

This chapter will uncover the implementation details of Stage 5: Global Relation Extraction in 6 aspects, similar to the previous chapters, i.e., the algorithms, prompts, assumptions made, limitations, complexity, and decisions made.

9.1 Algorithm

The general pattern for global relation extraction is to perform a pairwise check of any possible pair of entities and see if there is a potential relation between them, given the context of the whole document. However, checking every possible pair can be computationally expensive. Therefore, our algorithm only selects the top 10% relevant entities in the paper. Even so, $O((0.1 \cdot n)^2)$ can still scale badly for long paper. Therefore, we set the maximum number of entities involved in global relation extraction to be 20.

To obtain the relevancy of each entity so that they can be ranked, the algorithm compares the embedding between the entity and the paper.

Because a whole paper can potentially exceed the context limit of the LLM for both encoders and decoders, the algorithm first shrinks the document based on the assumption that the first parts and the last parts of a paper are always the most important parts. The goal of shrinking is to keep the document as original as possible while the document can fit the LLMs. Therefore, the algorithm prompts the LLM decoder to do summarisations starting from its middle sections and up to the abstract and conclusion. If it turns out that the updated paper is still too long, another iteration will be performed.

Since the updated paper can now fit by an LLM encoder or encoder, the embedding of the paper can be obtained by calling an LLM encoder. We have chosen the encoder M3 instead of BGE due to a large context limit of 8196 tokens. However, we used M3 for coreference resolution because BEG specialised in generating embedding of English text

9 Stage IV: Global Relation Extraction

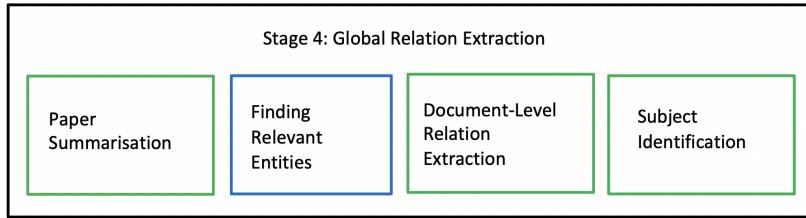


Figure 9.1: Stage 4: Global Relation Extraction

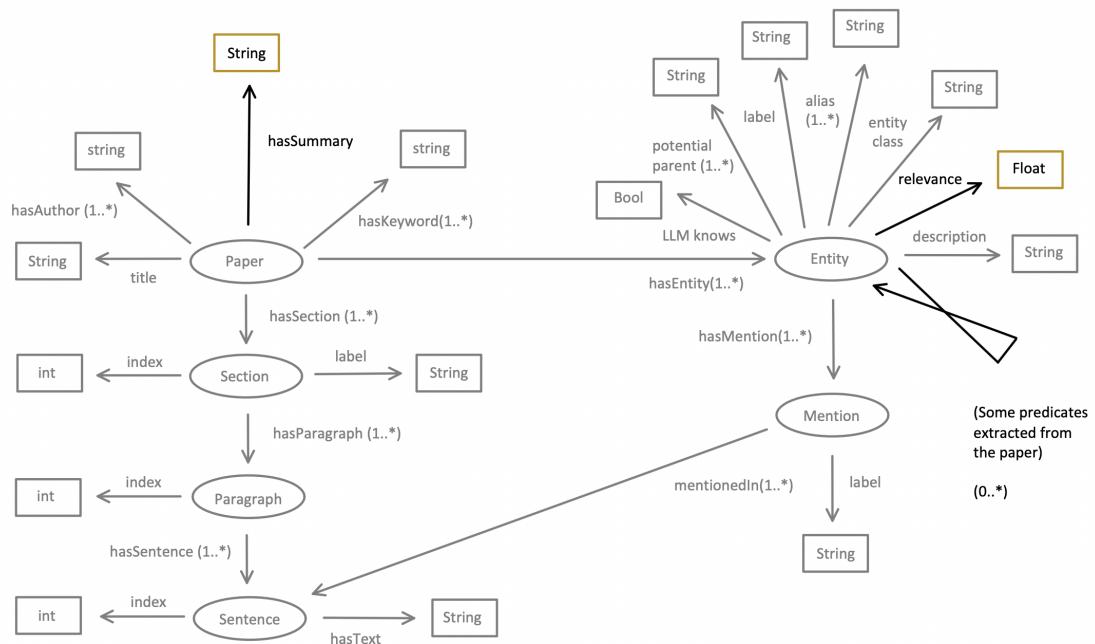


Figure 9.2: Input and Output

while M3 is multilingual, and our datasets are all in English.

For the substage of relation extraction, the algorithm loops through every possible pair within the most relevant entities. Within each iteration, the label, potential parent types, and description are given to the LLM. The paper's title, keywords, authors, and updated content with shorter length are also given. The output from the LLM is the most relevant predicate between them.

However, in practice, we found that even though the LLM can output a predicate between two entities, it cannot correctly identify the subject and object within the relation. Therefore, we perform another set of LLM calls, which give the label, potential parent types and description of both entities as well as their predicate. The LLM needs to identify the subject based on the given information.

9.1 Algorithm

The general picture of Stage 4 is shown in Figure 9.1, while the input and output are shown in Figure 9.2. The input is marked in grey, and the output is marked in colour. The algorithm of Stage 4 is as follows:

Algorithm

Input: An intermediate KG with some predicates connecting the entities.
Output: An intermediate KG with more predicates connecting the entities.

Paper Summarisation

While the paper's length is over the same token limit,

from the middle section to the first and the last section of the paper,

[LLaMA Prompt_1] Ask the LLM to summarise the section into half its original length.

If the updated paper is below the token limit,

break.

Finding Relevant Entities

[M3] Embed all the entities based on their label, potential parent types, and description.

[M3] Embed the updated paper

For each entity in the KG,

Compute the similarity score between the entity and the paper.

Store it as a new attribute under the entity.

Get the top 10% of the most relevant entities or the top 20 relevant entities from the KG into the list called relevant_entities.

Global Relation Extraction

For entity_1 in relevant_entities,

for entity_2 in relevant_entities,

[LLaMA Prompt_2] Using the paper's title, authors, keywords, and updated content, along with the label, description, and potential

9 Stage IV: Global Relation Extraction

parent types of both entities, ask the LLM to return at most one predicate between these two entities.

Subject Identification

For the newly extracted predicate of each relevant entity pair,

[LLaMA Prompt_3] Using the label, description, and potential parent types of both entities, plus the previously identified predicate, ask the LLM to determine the subject in this relation.

Tidy Up

Move all the triples to the paper level.

Return all redundant triples whose subject, predicate, and object are all the same.

9.2 Prompts

The following are the simplified versions of the three LLM prompts mentioned in the algorithm.

Prompt 1: Paper Summarisation

Task

You are a linguistic expert involved in a text summarisation task. Given the text in the **Input** section below, your job is to summarise it into half its original length while retaining key information and ideas.

Input

Output

Prompt 2: Global Relation Extraction

Task Definition

You are a linguistic expert involved in a relation extraction task for finding the relationship between the terms "{term_1}" and "{term_2}" in an academic paper. Given the title, authors, keywords, and content of the paper as well as the types and description of each term, your task is to extract the **most important** predicate that connects the term "{term_1}" and the term "{term_2}".

Output Format ...

```
### Paper
**Title**: {title}
**Keywords**: {keywords}
**Authors**: {authors}
**Content**: {content}

### Term 1: {term_1} ...
### Term 2: {term_2} ...
**Output**:
```

Prompt 3: Subject Identification

Task Definition

You are a linguistic expert involved in a subject identification task. Given the terms "{subject_label}" and "{object_label}" related by the predicate "{predicate}", your task is to determine the subject of this relation. Your answer should be a triple, providing the new orders for Term 1 and Term 2.

- Your answer should be based on the labels, types, and descriptions of both terms, the predicate, and your common knowledge.
- You should also check if the predicate is active or passive in order to determine the subject
- Ask yourself whether "{subject_label} {predicate} {object_label}" is more fluent or "{object_label} {predicate} {subject_label}" is more fluent and natural

```
## Output Format ...
## Input ...
## Output ...
```

9.3 Assumptions, Decision Made, and Limitations

During the implementation, we have assumed that the relations returned by the LLM are indeed the relations that exist in the document.

However, it was found that that was not always the case. The LLM we used, LLaMA, suffered seriously from hallucination issues in this stage. That is, for each relevant entity pair, no matter if they are truly related by some predicate or not, the LLM always return predicates, even though we have explicitly stated in the prompt that “at most one” predicate should be returned.

In fact, in the original implementation, we even do not have this “at most one” constraint. That is, we allow the LLM to return any number of predicates found in the document.

9 Stage IV: Global Relation Extraction

However, the finding is that only the first one or two predicates are usually the correct predicate in between the entities, while others are merely "filled in the numbers".

Therefore, we made the decision that only one predicate is allowed to be returned between any two entities. This decision allows the LLM to focus on returning the most likely predicate.

In the original implementation, there is also another layer, which, given the triple found during the global relation extract, asks the LLM again to check whether the relation between two entities exists in the paper. However, it turns out that even a simple 'Yes' or 'No' response can itself be a hallucination.

Therefore, we decided not to include such a layer to reduce the overall complexity. The method for reducing hallucination is based on restricting the number of predicate returns. And we assume that the only predicate return is truly a predicate in the paper.

Since there is only one predicate return, this leads to the limitation that if there indeed exists multiple predicates between some entities, the pipeline will miss most of them.

9.4 Complexity Analysis

The complexity of this stage can be modelled by $O(L + \text{Max}(20^2, (0.1 \cdot E)^2) + R_2)$, where L is the length of the paper, E is the number of entities, and R_2 is the number of triples extracted in Stage 4, the current stage only. During the summarisation, our algorithm loops through each section of the document and summarises each section individually. However, there is not an upper bound on how many times it loops through the whole paper. Since academic papers usually have a fixed length restricted by some academic conference and it does not go extremely long, it turns out that the summarisation can usually finish within one epoch (all the sections in the paper). Therefore, we model it as $O(L)$ for simplicity. For the global relation extraction part, this is either the top 20 entities or the top 10% entities. Therefore, it is $O(\text{Max}(20^2, (0.1 \cdot E)^2))$ since pairwise comparison is formed. In the end, for each triple found, the algorithm further confirms the subject and object, leading to an additional complexity of $O(R_2)$.

Chapter 10

Stage V: Schema Generation

This chapter will provide the implementation details about the second last stage, schema generation. It will also follow a similar layout to the previous chapter, i.e., the algorithms, prompts, assumptions made, limitations, complexity, and decisions made. This is also the last chapter about pipeline implementation because the last stage, KG post-processing, has been covered in Chapter 5, the pipeline overview.

10.1 Algorithm

There are two main parts for creating a concrete schema of the KG. One is to construct a taxonomy for the entities within the KG. The other is to normalise predicates.

The first part creates a hierarchy of supertype and subtype among all entities in the KG, e.g., (“ANU”, “has a broader term”, “Public University”) and (“Public University”, “has a broader term”, “University”) as long as all “ANU”, “Public University”, and “University” are the entity exist in the KG, which essential form a directed acyclic graph. In fact, some of the relationships may have already been found in Stage 3: Local Relation Extraction and Stage 4: Global Relation Extraction. However, this stage aims to make the KG more complete by only focusing on the hierarchical relations.

The other is to merge predicates referring to the same meaning so the redundancy of predicates can be reduced, which can further reduce the number of redundant triples, like (“ANU”, “located in”, “Canberra”) and (“ANU”, “is in”, “Canberra”).

The high-level pattern of the algorithm of this stage is similar to Stage 2: Entity Disambiguation and Coreference Resolution. In the taxonomy part, the essential goal is to find a matching parent entity in the KG for each potential parent of the entities so that an edge, “has a broader term” or “skos:broader” is drawn between an entity and its parent entity. Before this stage, the potential parent types are not entities but strings,

10 Stage V: Schema Generation

and this stage matches some of them to the actual entities in the KG.

The algorithm starts with gathering the potential parents of all entities into a Python Dictionary, where the key is the name of the potential parent. We made an assumption that the potential parent types are unambiguous. This is because entity types are usually general concepts, and the fact that they are general makes them unlikely to have too many distinct meanings. This is because their meanings are already broad. This assumption is essentially the same as one of the assumptions made during the Entity Disambiguation in Stage 2. With this parent-type dictionary. The algorithm prompts the LLM to generate a description of each type without any assistance context. Since they are general, we assume that the LLM knows all of them. After that, we embed these parent types using their labels and descriptions, as well as embedding all the entities in the KG. For each entity and for each potential parent type, the algorithm finds the embedding of the parent type by checking the parent-type dictionary. And compare it with all the other actual entities in the KG, if it turns out that a potential parent of an entity is highly similar to an entity on the KG according to their embedding, we create a “skos:broader” relationship between these two entities. After the taxonomy graph is created, we perform a basic consistency check by removing any loop in the directed acyclic graph of taxonomy using the Depth-First Search Algorithm ([Tarjan, 1972](#)).

As for predicate normalisation, this is highly similar to coreference resolution. However, we do not simply merge two predicates only based on their label, like when merging named entities and general concepts because the meaning of a predicate can vary a lot depending on the context, like (“Amy”, “run”, “a business”) and (“Amy”, “run”, “a marathon”). Therefore, we perform a more conservative approach. That is, first, generate a description for the predicate based on the triple, then embed the predicate and finally merge the predicate if their embeddings are found similar.

After the predicate normalisation, the number of predicates is fixed. Then we assign each predicate an IRI.

In the end, we eliminate any repeating triple after the normalisation where all its subject IRI, object IRI, predicate IRI are the same.

The general picture of Stage 4 is shown in Figure 10.1, while the input and output are shown in Figure 10.2. The input is marked in grey, and the output is marked in colour. The ontology of the KG does not change, but only more relations between entities are created, and the number of predicates is reduced.

The algorithm of Stage 4 is as follows:

Algorithm

Input: An intermediate KG with some predicates and some (entity, predicate, entity) triples

Output: An intermediate KG with less predicates and more

10.1 Algorithm

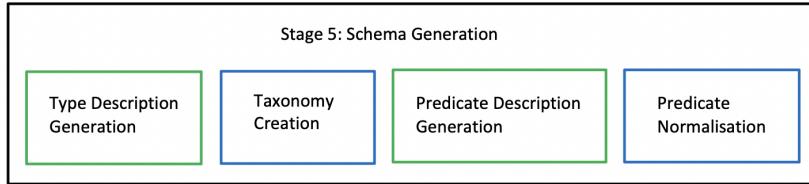


Figure 10.1: Stage 5: Schema Generation

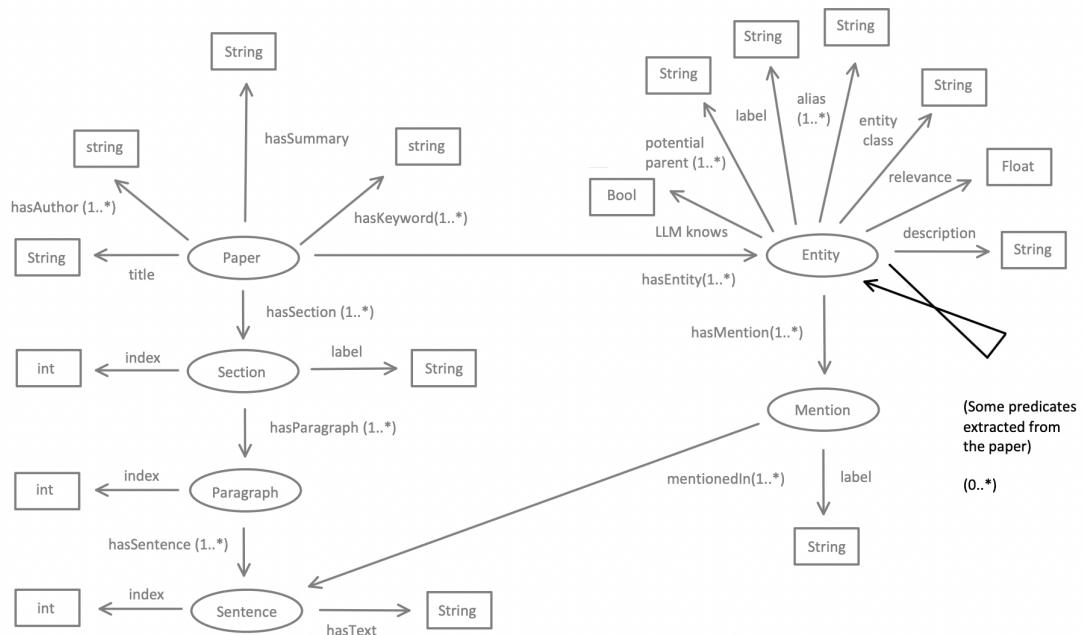


Figure 10.2: Input and Output

10 Stage V: Schema Generation

```
(entity, predicate, entity) triples

# Gathering Types

Create a dictionary for storing potential parent types for all entities,
named as type_dict.

For each entity in the KG,
    for each potential parent type of the entity,
        add it into the type_dict.

# Generating Type Description

For each type in type_dict,
    [LLaMA Prompt_1] asks the LLM to generate a general description based
    on its common knowledge.

# Embed Types and Entities

[BGE] Embed all types in type_dict based on their label and description
[BGE] Embed all entities in the KG based on their label and description

# Finding Parents

For entity_1 in the KG,
    for each potential parent type of the entity,
        get the embedding of the potential parent type from type_dict.

    For entity_2 in the KG,
        compute the cosine similarity between the embedding of the
        potential parent type and the embedding of entity_2
        if the similarity > 0.95,
            draw a temporary and directed edge between entity_1 and
            entity_2.
```

```
# Removing Cycles
```

For the subgraph created by the temporary links, use the Depth-First Search algorithm to remove all cycles.

Put the updated temporary links into the KG.

```
# Predicate Normalisation
```

Now the KG has a list (set) of (Entity IRI, predicate name, Entity IRI) triples.

```
# Generate Descriptions for Predicates
```

For each triple in the KG,

[LLaMA Prompt_2] Given the triple, ask the LLM to generate a general description for the predicates.

Store the description directly after the triple. (essential making it into 4-tuple)

```
# Embed Predicate
```

[BGE] Embed the predicate of all triples based on their labels.

Store the embedding directly after the triple. (essential making it into 5-tuple)

```
# Finding relevant predicates
```

For triple_1 in the KG,

for triple_2 in the KG,

get the embeddings of the predicates of triple_1 and triple_2.

Compute their cosine similarity.

If their cosine similarity is over 0.95,

draw a temporary link between these two triples.

10 Stage V: Schema Generation

```
# Predicate Resolution

For each triple in the KG,

    pull out their predicate into a graph, as well as the temporary links
    between these predicates.

    Maintain a mapping between the predicate in the triple and the node in
    the predicate graph.

# Up to now, all predicates are still distinct nodes.

Use the modified Bron-Kerbosch algorithm to find all
the disjoint maximal cliques in the predicate graph.

For each clique of predicates (including those with only one node),

    group them into a single predicate.

    Choose any of the names of the predicate as a representative name for
    the final predicate.

    Assign a unique IRI for the predicate.

For each triple in the KG,
    change it from (Entity IRI, predicate name, Entity IRI) into
    (Entity IRI, Predicate IRI, Entity IRI) using the mapping created
    before.

Remove any redundant predicates with the same IRIs in all positions.
```

10.2 Prompts

The following are the simplified versions of the three LLM prompts mentioned in the algorithm.

Prompt 1: Type Description Generation

```
## Task
```

```
You are a linguistic expert involved in a description
writing task for a term "{label}" found in a paper.
Given the term and based on your background knowledge
about the term, your task is to write a single-
sentence description that can precisely tell what the
actual entity or concept the term "{label}" refers to.
```

```
## Output Format ...
## Input ...
## Output ...
```

Prompt 2: Predicate Description Generation

```
## Task
You are a linguistic expert involved in writing a
general description for the predicate "{predicate}" in
the (subject, predicate, object) triples like "
({subject}, {predicate}, {object})".
```

- Your description should be general and not tied to any specific instance of the given triple.
- The provided triple is only there to help you understand the use of the predicate.
- Your description should explain how the predicate connects the subject and object in a general sense.

```
## Output Format ...
## Input ...
## Output ...
```

10.3 Assumptions Made

The following is the summary of the key assumptions made in this stage, some of which have been mentioned earlier.

1. When grouping the potential parent types into a dictionary based on their names, we assume that these types are ambiguous. That is, their labels are sufficient to represent themselves.
2. When generating descriptions for all potential parent types, we assume that the LLM has all background knowledge about the types, similar to generating descriptions of predicates.

10.4 Limitations

Several limitations can be found in this stage:

1. The potential parent types are created at the very first stage of the pipeline before coreference resolution. Therefore, these potential parent types are created based on the mention of entities within a limited context only. Therefore, they may not be highly accurate. A potential way to reduce this limitation is to type any entity after the Coreference Resolution stage. However, it may introduce extra complexity.

10 Stage V: Schema Generation

2. When generating descriptions for types and predicates, the description may not truly represent the types and predicates, i.e. they can be lossy.
3. When generating embedding for types and predicates, the embedding may not truly represent the types and predicates, i.e. they can be lossy. A more advanced technique for generating embedding for building taxonomy is to create box embedding ([Onoe et al., 2021](#)) embedding for entities in the KG. In this way, if the embedding of one entity is involved in the box embedding, then we can know that one is the parent entity of the other.
4. When determining the parent entity of an entity and checking if the two predicates are similar, we use the cosine similarity between their embedding. However, the cosine similarity does not tell the length of these two vectors. Therefore the result can be lossy.
5. After computing the cosine similarity, we create a temporary link between the entity and its parent entity or between two predicates as long as their similarity is above some threshold with further double-checking, unlike the Coreference Resolution stage. This is because we tend to improve the efficiency of the pipeline. However, such a more aggressive way of linking entities or predicates can lead to more errors.
6. When merging a group of predicates into a single predicate, the algorithm does not have an extra step and heuristic to find the most representative name for the predicates for simplicity.

10.5 Complexity Analysis

The complexity of this stage can be modelled as $O(E + R)$, where E is the number of entities and R is the total number of predicates (= triples) of the relations of the entities after the local and global relation extraction. Although an entity normally has several potential entity types, some potential entity types are shared among different entities, e.g., the “ANU” and “USYD” are all “Universities”. Therefore, for simplicity, we roughly model it as $O(E)$. Since the algorithm also call the LLM decoder to generate a description for the predicate of each triple, there is also a complexity of $O(E)$, which results in an overall complexity of $O(E + R)$.

10.6 Design Decision

Some decisions made in this stage are similar to Stage 3: Entity Disambiguation and Coreference Resolution. They include the use of description for generating embedding and the use of disjoint maximal cliques to group relevant predicates.

However, the major difference is the removal of the double-checking mechanism when two of the potential parents of an entity are found similar to another entity, or two

10.6 Design Decision

predicates are found similar. This decision may reduce complexity but potentially result in lower precision. However, the recall may also increase because the double-checking mechanism itself is error-prone, and LLM can be too conservative. Therefore, this is a trade-off. Eventually, we increase the threshold from 0.9 to 0.95 to prioritise precision over recall.