Comments on Compendium-Text (after creating Themenbaumstruktur)

Nghia Duong-Trung

23.03.2025

1 Overview of Requirements

1. Start with a hierarchical topic tree (in JSON) that may or may not contain detailed textual content for each node.
2. Generate a text-based compendium that succinctly summarizes the essential concepts at each node or category.
3. Optionally enrich the text with external references using NER/NEL, linking key entities to sources like Wikipedia, Wikidata, or DBPedia.
4. Allow for different usage scenarios, from *empty* (or minimal) collection descriptions to *content-rich* collections.

1.1 Example JSON Reference

Below is a *minimal* JSON structure representing a topic node:

Listing 1: Minimal JSON for a Topic Node

```
{
  "general": {
    "identifier": "UniqueID-123",
    "title": "Descriptive Title",
    "language": "en",
    "description": "Short definition or summary text.",
    "keyword": ["Keyword1", "Keyword2"]
  },
  "classification": [
    {
      "purpose": "discipline",
      "taxonPath": [
        {
          "source": "MyTopicHierarchy",
          "taxon": [
            {
              "id": "1",
              "entry": "Main Category"
            },
            {
```

```
            "id": "1.1",
            "entry": "Subcategory"
          }
        ]
      }
    ]
  }
 ]
}
```

While simplified, such a structure provides a basis for the compendium generation process: we have `title`, `description`, and `keyword` fields, plus classification information that can orient hierarchical relationships.

2 Possible Solutions for Compendium Generation

I present three major approaches, each suitable for a specific content scenario:

2.1 Single-Step Generation (Quick Overview)

1. **Gather Metadata:** Parse the JSON to collect `title`, `description`, and `keyword` data for each node in the hierarchy.
2. **One-Pass Prompt:** Provide these fields to a LLM with instructions such as:

   ```
   "Please summarize the hierarchy into a
   cohesive compendium text. Include brief
   descriptions and relevant keyword expansions."
   ```

3. **Optional NER/NEL:** In the same prompt or a secondary pass, request the model to identify entities and link them to external sources (e.g., Wikipedia).

Pros:

- Fast, low-complexity pipeline.
- Easy to implement if the data engineers have a single LLM call in mind.

Cons:

- Limited control over depth of content.
- Single-step approach may lead to inaccuracies in entity linking, especially if the text is very short.

2.2 Iterative Generation with Text Expansion (Use Case 2)

Scenario: The collections are nearly empty, holding only short descriptions (2–3 lines) and a few keywords.

1. **Generate Draft Text:** Prompt an LLM to produce a more extensive *draft* (1–2 pages) for each node, based on available metadata (e.g., educational level, subject) and the node's title/keywords. This text serves as a placeholder or synthetic content, providing enough substance for deeper analysis.
2. **Run NER/NEL:** Feed the newly generated text to an NER system. Identify key terms (people, organizations, scientific concepts, etc.), then link them to external knowledge sources such as Wikipedia. This can be done in:
    - A *single LLM pass* if your model is capable of both recognition and linking.
    - A *two-stage pipeline* using a specialized NER service (e.g., IBM zshot)[1][2] and a subsequent linking step.
3. **Refinement:** Optionally, ask a second prompt to "merge the recognized entities with the original text" to create a polished compendium section.
4. **Compile Compendium:** Combine the refined text for each node into a single document. This final step can also *format* or *structure* the text for easy reading.

Pros:

- Produces sufficiently large text for robust NER/NEL.
- More accurate linking due to improved context.

Cons:

- Multi-step pipeline with potential overhead in orchestration.
- Synthetic text might deviate from real-world context if the prompt is poorly designed or domain knowledge is lacking.

2.3 Using Existing Collection Content (Use Case 3)

Scenario: The topic tree already contains references to actual documents or detailed descriptions (e.g., PDF files, extended textual data).

1. **Content Aggregation:** For each node, gather relevant text from the documents. If the node has 3–4 PDF references, you might combine or summarize them to produce a single consolidated text body.

[1]https://ibm.github.io/zshot/
[2]https://research.ibm.com/publications/zshot-an-open-source-framework-for-zero-shot-learning

2. **LLM Summaries:** Prompt the LLM to create a 1–2 page summary that highlights key points from the aggregated text, referencing the node's `title`, `description`, and `keywords` as a guide for structuring the summary.
3. **NER/NEL:** Extract entities from this summary and link them to external knowledge sources.
4. **Compile Final Compendium:** Output a multi-section compendium that merges the newly summarized text for each node with consistent formatting.

Pros:

- High factual accuracy, as the text is *based on real content*.
- Reduces risk of "hallucination" or misinformation in the final compendium.

Cons:

- Requires ingestion and partial summarization of potentially large documents.
- Additional data-engineering overhead to parse or chunk PDFs if they are unstructured.

3 Implementation Considerations

3.1 Data Pipeline Orchestration

Data engineers may want to implement these steps in a pipeline architecture, using tools like **Apache Airflow** or **Luigi**. A pipeline approach allows the process of generating compendium texts to be:

- **Modularized:** Each step (e.g., reading JSON, invoking an LLM, running NER/NEL, merging final results) can be written as a distinct task.
- **Fault-Tolerant:** If one step fails (e.g., NER times out), you can retry from that step without restarting the entire workflow.
- **Scheduled & Monitored:** Pipelines can be triggered automatically (e.g., nightly) or on demand, with real-time logging and alerting.
- **Scalable:** A well-structured pipeline can run tasks in parallel, handle large volumes of data, and adapt to changing loads.

3.1.1 Apache Airflow

Apache Airflow[3][4] is an open-source platform to programmatically author, schedule, and monitor workflows. It uses a concept called *Directed Acyclic Graphs* (DAGs) to represent the flow of tasks. Data engineers write Python code to define these DAGs, and tasks can be orchestrated across multiple servers or containers.

[3] `https://airflow.apache.org/`
[4] `https://github.com/apache/airflow`

Key Airflow Concepts:

- **DAG:** A pipeline, composed of operators that run in a sequence with dependencies.
- **Operator:** A single task, such as "Call LLM for text generation" or "Perform NER/NEL on expanded text." Airflow includes many built-in operators for data transfer, Bash commands, Python functions, etc.
- **Scheduler:** Schedules DAGs based on time or external triggers. Monitors the running tasks and handles retries or failure scenarios.
- **Web UI:** An interface to visualize DAG runs, check logs, and re-run failed tasks.

Example Airflow DAG Pseudocode:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

default_args = {
    'owner': 'data-engineer',
    'depends_on_past': False,
    'start_date': datetime(2023, 1, 1),
    'retries': 1
}

with DAG('compendium_generation_dag',
         default_args=default_args,
         schedule_interval='@daily') as dag:

    def fetch_topic_tree(**context):
        # Step 1: Ingest JSON structure
        # ...

    def generate_draft_text(**context):
        # Step 2: LLM-based text generation
        # ...

    def run_ner_nel(**context):
        # Step 3: Named Entity Recognition & Linking
        # ...

    def merge_and_store(**context):
        # Step 4: Merge results, store compendium
        # ...

    t1 = PythonOperator(
        task_id='fetch_topic_tree_task',
        python_callable=fetch_topic_tree
    )

    t2 = PythonOperator(
        task_id='generate_draft_text_task',
        python_callable=generate_draft_text
    )

    t3 = PythonOperator(
        task_id='run_ner_nel_task',
        python_callable=run_ner_nel
    )

    t4 = PythonOperator(
        task_id='merge_and_store_task',
        python_callable=merge_and_store
    )

    t1 >> t2 >> t3 >> t4
```

3.1.2 Luigi

Luigi[5][6], developed initially by Spotify, is another open-source Python package for building complex pipelines of batch jobs. Luigi structures tasks similarly, but uses a *dependency graph* approach through Python classes.

Key Luigi Concepts:

- **Task:** A unit of work, which can specify its dependencies (other tasks).
- **Target:** Typically a file or data location that the task produces.
- **Scheduler:** Coordinates running tasks based on declared dependencies.

Example Luigi Task Pseudocode:

```
import luigi

class FetchTopicTree(luigi.Task):
    def output(self):
        return luigi.LocalTarget('topic_tree.json')

    def run(self):
        # Ingest JSON data
        # Write to self.output()

class GenerateDraftText(luigi.Task):
    def requires(self):
        return FetchTopicTree()

    def output(self):
        return luigi.LocalTarget('draft_text.json')

    def run(self):
        # LLM-based text generation
        # Write to self.output()
```

[5]https://github.com/spotify/luigi
[6]https://luigi.readthedocs.io/en/stable/

```
class RunNERNEL(luigi.Task):
    def requires(self):
        return GenerateDraftText()

    def output(self):
        return luigi.LocalTarget('ner_nel_results.json')

    def run(self):
        # NER/NEL
        # Write results

class MergeAndStore(luigi.Task):
    def requires(self):
        return RunNERNEL()

    def output(self):
        return luigi.LocalTarget('final_compendium.json')

    def run(self):
        # Merge results, produce final compendium
        # Write to self.output()

if __name__ == '__main__':
    luigi.run()
```

3.2 Handling Large Documents

When dealing with big PDFs:

- **Chunking Strategy:** Split documents into manageable segments (e.g., a few thousand tokens each) and summarize them iteratively.
- **Vector Databases:** For advanced context retrieval, data engineers might store vector embeddings (via e.g., **FAISS** or **Milvus**) and query relevant chunks before summarization.

3.3 Quality Control and Feedback Loops

To ensure reliability:

- **SME Review:** Subject Matter Experts can provide feedback on the compendium text, verifying correctness and completeness.
- **Versioning:** Maintain versioned records of compendium text to allow rollbacks or incremental improvements.
- **Automated QA Checks:** If the domain is highly standardized, you can compare the LLM-generated text against known data or ontologies to detect anomalies.

3.4 Data Storage and Export

Once the compendium text is generated:

- **Re-Embed in JSON:** Create a new field, such as `"compendiumText": "..."` within each node's metadata.
- **Separate Documents:** For complex projects, store the final text in a robust content management system, referencing node identifiers or classification paths to maintain traceability.

4 Architectural Patterns and Best Practices

4.1 Modular Task Definition

Each major step of the compendium pipeline should be defined as a stand-alone task or operator:

- **JSON Ingestion:** Reading and parsing the topic tree from an external source.
- **Content Retrieval:** (Optional) Summaries from associated PDFs or documents.
- **Draft Text Generation:** Large Language Model calls for text expansion or compendium drafting.
- **NER/NEL:** Named Entity Recognition and Linking, possibly split into two tasks if the pipeline is large (one for NER, one for NEL).
- **Compilation/Storage:** Merging results and saving the final compendium text into a designated storage location (database, S3, etc.).

4.2 Parallelism and Scaling

- If the topic tree has many nodes, each node's text generation and NER/NEL could run in parallel.
- Tools like Airflow and Luigi enable concurrency settings, so you can scale out horizontally if tasks are CPU-intensive or need separate LLM queries.

4.3 Logging and Monitoring

- **Centralized Logs:** Both Airflow and Luigi capture detailed logs for each task, which are critical for debugging or audits.
- **Monitoring/Alerting:** Set up email or Slack alerts on pipeline failure or retry events. This ensures that data engineers quickly identify and resolve issues.

4.4 Reusability and Incremental Updates

- You may only want to rebuild a compendium for updated or new nodes rather than reprocessing the entire tree.
- Use *incremental* task logic, where existing outputs are treated as up-to-date unless there is a change in the input data source. This is especially convenient in Luigi, where tasks check if their `output` targets already exist.

References