SemLink Documentation

Contents

1 Prerequisites: Setting up the Conda Environment

This pipeline requires a dedicated Python environment with a specific set of libraries. The easiest way to manage this is with Conda. If you don't have Conda installed, you can follow the steps below.

1.1 Install Miniconda

Miniconda is a minimal installer for Conda, Python, and their core dependencies. It is the recommended way to get started.

1.1.1 For Linux

Use `wget` or `curl` to download the installer script, then run it.

```
# Download the latest Linux installer
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
    -O miniconda.sh

# Run the installer in silent mode (-b) and force a specific prefix (-p)
bash miniconda.sh -b -p $HOME/miniconda3

# Remove the installer script
rm miniconda.sh

# Initialize Conda for your shell (e.g., bash or zsh)
~/miniconda3/bin/conda init
```

1.1.2 For macOS

Use `curl` to download the installer script, then run it. For Apple Silicon Macs, use the `arm64` installer. For Intel-based Macs, use the `x86_64` installer.

```
# For Apple Silicon (M1/M2/M3)
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-arm64.sh
    -o miniconda.sh

# For Intel-based Macs
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
     -o miniconda.sh

# Run the installer in silent mode (-b) and force a specific prefix (-p)
bash miniconda.sh -b -p $HOME/miniconda3

# Remove the installer script
rm miniconda.sh

# Initialize Conda for your shell (e.g., zsh)
~/miniconda3/bin/conda init zsh
```

After running `conda init`, you may need to close and reopen your terminal for the changes to take effect. You should see `(base)` at the beginning of your prompt, indicating the base Conda environment is active.

1.1.3 For Windows

```
# PowerShell
Invoke-WebRequest -Uri "https://repo.anaconda.com/miniconda/Miniconda3-
    latest-Windows-x86_64.exe" -OutFile "miniconda.exe"
.\miniconda.exe /InstallationType=JustMe /AddToPath=1 /S /D=%UserProfile%\
    miniconda3
```

1.2 Create and Activate the Environment

Once Miniconda is installed, navigate to the directory containing your `environment.yml` file and run the following command to create the environment. This will install all the required packages listed in the file.

```
conda env create -f environment.yml
```

Once the environment is created, activate it with the following command:

```
conda activate semlink
```

If using Visual Studio Code, after opening the python notebook, select semlink as python kernel to execute the code.

1.3 Verify the Installation

To ensure everything is installed correctly, you can list the packages in your new environment:

```
conda list
```

2 Pipeline Overview

This pipeline offers a structured way to load raw data, semantically annotate it, and discover potential join relationships. It uses a series of modules to transform raw data, enrich it with meaningful metadata, and create a graph-based model of its relationships.

2.1 Important notes

The algorithms presented use the OpenAI API to generate column descriptions, annotations and more. An OpenAI API key is required. Create and .env file to store your API key as follow:

```
OPENAI_API_KEY="your-api-key"
```

and save the .env file in the same directory that contains the src folder.

The **ArcheType** module needs to be downloaded from its repository, unpacked and placed in the src directory.

2.2 Overall Pipeline Flow

The pipeline consists of four main steps, executed in order to achieve the final output:

1. **Data Loading and Preprocessing:** The `data_loader.py` module takes a directory of CSV/TSV files and converts them into a standardized JSON format. It performs basic statistical analysis and, if a Large Language Model (LLM) is used, generates initial descriptions.
2. **Schema Generation:** The `schema_generator.py` module uses the processed data to generate a LinkML schema. This schema acts as a conceptual model, defining classes and attributes that represent the data. This step also leverages an LLM to suggest a schema based on column names and descriptions.
3. **Semantic Annotation:** The `semantic_annotation.py` module takes the processed data and the generated LinkML schema to semantically annotate each column. It maps each column to a specific class and attribute within the schema.
4. **Join Discovery and Graph Generation:** The final step, handled by `join_discoverer.py`, uses the semantic annotations to find potential joinable columns. It generates embeddings for each column and computes similarity metrics. The final output is a set of CSV files for use in a graph database like Neo4j, representing a graph of data nodes and their relationships.

2.3 Step 1: Data Loading and Preprocessing (`data_loader.py`)

This is the starting point of the pipeline. It handles ingesting your raw data and generating an initial structured representation.

- **Input:**
  - A directory containing `.csv` or `.tsv` files.
  - (Optional) A directory containing `.json` metadata files corresponding to the `.csv` files.
  - (Optional) An initialized OpenAI client for LLM-based descriptions.
- **Input Data Structure:**
  - The pipeline expects semi-structured data in CSV or TSV format.
  - Each file is treated as a single table, with column headers used to identify columns.
- **Output:**
  - A JSON file named `data_lake.json`.
  - The output is a list of dictionaries, where each dictionary represents a file (table) and contains metadata about the file and its columns. This includes basic statistics and, if an LLM is used, high-level descriptions.

Here's how to run this step:

```
# Import the necessary class
from data_loader import DataLoader
from openai import OpenAI

# Initialize OpenAI client and DataLoader
client = OpenAI(api_key="<YOUR_OPENAI_API_KEY>")
loader = DataLoader(openai_client=client)

# Run the data loading and processing step
data_lake = loader.load_and_describe_datalake(
    directory='<YOUR_DATA_DIRECTORY_PATH>',
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)
```

2.4 Step 2: Schema Generation (`schema_generator.py`)

This step creates a conceptual schema from your data, which is essential for semantic annotation.

- **Input:**
  - The `data_lake.json` file generated in Step 1.
  - An initialized OpenAI client.
  - (Optional) A pre-existing LinkML schema file (`.yaml`) for pruning only.
- **Input Data Structure:**
  - The primary input is the `data_lake.json`. The module uses the `llm_description` and `llm_column_description` fields to inform the schema generation.
- **Output:**
  - A LinkML schema file named `linkml_schema.yaml`.
  - A pruned version of the schema named `linkml_schema_pruned.yaml`.
  - This schema defines classes (entities) and attributes (their properties), providing a standardized vocabulary.

Here's how to run this step:

```
# Import the necessary class
from schema_generator import SchemaGenerator

# Initialize SchemaGenerator
schema_gen = SchemaGenerator(openai_client=client)

# Run the schema generation step
linkml_schema = schema_gen.generate_schema_from_datalake(
    data_lake=data_lake,
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)

# Prune the schema
pruned_schema = schema_gen.prune_schema(
    yaml_schema=linkml_schema,
    data_lake=data_lake,
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)
```

2.5 Step 3: Semantic Annotation (`semantic_annotation.py`)

This step enriches your data by mapping it to the generated schema, adding a semantic layer that describes the meaning of each column.

- **Input:**
  - The `data_lake.json` file from Step 1.
  - The `linkml_schema_pruned.yaml` file from Step 2.
- **Input Data Structure:**
  - The module takes the `data_lake.json` and uses the column names and sample values from each file.
  - The `linkml_schema_pruned.yaml` file provides the target vocabulary for annotation.
- **Output:**
  - An annotated JSON file named `data_lake_annotated.json`.
  - The output is the same as `data_lake.json`, but with a new `semantic_annotation` field for each column, specifying its corresponding `class.attribute` from the LinkML schema.

Here's how to run this step:

```
# Import the necessary function
from semantic_annotation import archetype_annotation

# Run the semantic annotation step
data_lake_annotated = archetype_annotation(
    data_lake_list=data_lake,
    yaml_schema=pruned_schema,
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)
```

2.6 Step 4: Join Discovery and Graph Generation (`join_discoverer.py`)

The final step analyzes the semantically annotated data to identify and visualize potential join points.

- **Input:**
  - The `data_lake_annotated.json` file from Step 3.
  - (Optional) A pre-computed embeddings JSON file to skip the embedding generation.
  - An initialized OpenAI client.
- **Input Data Structure:**
  - The module requires the `semantic_annotation` field in the input JSON to be populated. It uses this information to generate unique identifiers for each column.
  - The OpenAI client generates vector embeddings for each column's context, which are used for similarity calculations.
- **Output:**
  - A JSON file named `embeddings.json` containing the vector embeddings for each column.
  - A set of CSV files (`distances.csv`, `nodes.csv`, and `edges.csv`) for importing into a graph database.
  - `nodes.csv`: Represents each column as a node.
  - `edges.csv`: Represents discovered join relationships as edges between nodes, with a `cosine_similarity` property.
  - `distances.csv`: A detailed breakdown of all computed similarity and distance values.

Here's how to run this step:

```
# Import the necessary class
from join_discoverer import JoinDiscoverer

# Initialize JoinDiscoverer
join_dis = JoinDiscoverer(openai_client=client)

# Generate embeddings
embeddings = join_dis.generate_embeddings(
    data_lake_annotated=data_lake_annotated,
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)

# Compute distances and export to Neo4j-compatible CSVs
_, _, _ = join_dis.compute_distances_and_export_neo4j(
    embeddings=embeddings,
    cosine_sim_threshold=0.8,
    anns_threshold=1.5,
    output_directory='<YOUR_OUTPUT_DIRECTORY_PATH>'
)
```

3 Datasets

- Valentine
- PheKnowLator benchmark KG
- Eurostat

4 Related Repositories

ArcheType – Module for semantic annotation. To download and place in the Src folder.