UASFRA-MS-PROJDIGI

Project Digitalization

Winter Semester 2023/24

Documentation
Working Group 4

M.Sc. Computer Science

Frankfurt University of Applied Sciences

Prof. Dr. Martin Simon

Project members:

Priya Singh - 1428461

Christopher Unkart - 1249284

Rainer Gogel - 1272442

Content

1. General Description

I. Overview

UASFRA-MS-PROJDIGI is a data science project and part of the requirements for the Master program (M.Sc.) in Computer Science at the Frankfurt University of Applied Sciences.

The project's goal is to create a NEO4J Knowledge Graph ("KG") populated with ESG data required to be reported by companies due to the European Sustainability Reporting Standards (ESRS) legislation.

The programs presented in here are able to create and query such a NEO4J Knowledge Graph using Python.

II. Project structure

All programs can be executed from the "main.py"-script in the root folder of this project. The "main.py"-script makes use of the following modules in the "src"-folder:

main.py

src

A_read_xbrl.py: Converts company's XBRL-files into JSON-files to later import the data into the KG

B_rdf_graph.py: Creates cypher queries based on the provided ontology.ttl-file and constructs the KG schema

C_read_data.py: Creates templates for importing the data from the JSON-files created earlier

D_graph_construction: Imports data from the JSON-files into the KG and loads addional data from wikidata and dbpedia

E_embeddings.py: Converts text of some Node's text properties into LLM embeddings to later do similarity search

F_graph_bot.py: Formulates questions in relation to data in the KG in human-readable form for a KB bot to answer them

G_graph_queries.py: Formulates questions in relation to data in the KG and gets results from Python functions

III. Settings and Installation

To run the functions in "main.py", some settings must be adjusted and NEO4J and related software needs to be installed first.

A. Project settings

In the root folder of this project, there is a "settings.py"-file with:

```
# PATHS
path_base = pathlib.Path("C:/your/path/to/the/root-folder/")
path_data = pathlib.Path(path_base, "src/data/")
path_models = pathlib.Path(path_base, "src/models/")
path_ontos = pathlib.Path(path_models, "Ontologies")
```

Only adjust the "path_base"-value to the path where this project (root folder) is located on your system. Leave all other paths untouched unless you want to change the location of these folders.

B. NEO4J database

There are different options to install the NEO4J database on your system. We recommend choosing the

Graph Database Self-Managed / NEO4J Server (Community or Enterprise)

version. Please follow the installation instructions here:

- Check the system requirements to install NEO4J: NEO4J system requirements
- Linux installation instructions: NEO4J Linux installation
- Windows installation instructions: NEO4J Windows installation

C. NEO4J plugins

Please make sure to also install the following NEO4J plugins:

- NEO4J neosemantics (or "n10s")
- NEO4J apoc
- NEO4J graph-data-science (or "gds")

Under Windows, the respective jar-files need to be downloaded and put into the "NEO4J_HOME/plugins" sub-folder of the "NEO4J_HOME"-folder on your system. Some "$NEO4J_HOME/conf"-files need to be adjusted. Please refer to the installation instructions here:

- NEO4J neosemantics installation instructions
- NEO4J apoc installation instructions
- NEO4J graph-data-science installation instructions

D. Python libraries

To use the programs, some Python libraries need to be installed first. Please install (i.e. with: **pip install ...**) all the libraries listed under [packages] in the Pipfile of the root folder:

```
Pipfile:
    [packages]
    neo4j
    pandas
    python-dotenv
    etc.
```

E. Check installations and settings

To check if the NEO4J installation succeeded and if Python NEO4J scripts can be executed:

1. Make sure you have adjusted the "$NEO4J_HOME/conf"-files as described in the NEO4J plugins installation instructions.
2. Open http://localhost:7474 in your web browser.
3. Connect using the username "neo4j" with the default password "neo4j". You might be prompted to change your password.
4. Go to the file "secrets_template.env" in the root folder of this project. Change the name of this file from "secrets_template.env" to "secrets.env". Set the NEO4J username and password from the web browser as your "NEO4J_USER" and "NEO4J_PW" there. You might also want to

insert your "OPENAI_API_KEY" there if you want to use the graph bot later.

5. To see if NEO4J and its plugins were installed correctly and can be used in "main.py", please run the following "test_installation.py"-script in the root folder:

test_installation.py:

```
""" This script's purpose is to check if the installation of NEO4J and the import of Python libraries succeeded."""

from src.G_graph_queries import GraphQueries

gq = GraphQueries()
df = gq._query_df(query="SHOW functions")

apoc = df.name.str.startswith('apoc').any()
n10s = df.name.str.startswith('n10s').any()
gds = df.name.str.startswith('gds').any()

if __name__ == '__main__':
    print(f"""INSTALLATIONS:
    apoc: {apoc}
    n10s: {n10s}
    graph-data-science: {gds}""")
```

6. You should now see "True" printed for all three prefixes:
    1. n10s.* (for neosemantics functions)
    2. apoc.* (for apoc functions)
    3. gds.* (for graph-data-science functions)

7. If you get an error or any of these three prefixes is missing ("False" in the printout), please go back and check/redo the settings and the installation.

IV. Usage

Make sure to have satisfied all requirements and adjusted all the settings as laid out above.

A. Functions

From the "main.py"-file in the root folder, you can now run the following functions by uncommenting the # CODE BLOCK below the desired function description:

main.py

```
0. Read XBRL-file into JSON-file. Please see: README-data.md-file.
    # CODE BLOCK
1. Load ontology and show schema of knowledge graph in browser. Please see: README-models.md-file.
    # CODE BLOCK
2. Load JSON-files/Company data into the NEO4J Knowledge-Graph. Please see: README-data.md-file.
    # CODE BLOCK
3. Enrich NEO4J Knowledge-Graph with external data from wikidata.
    # CODE BLOCK
4. Enrich NEO4J Knowledge-Graph with external data from dbpedia.
    # CODE BLOCK
5. Create text embedding for one of the text properties.
```

```
    # CODE BLOCK
6. GraphBot: RAG (Retrieval Augmented Generation) with NEO4J Graph.
    # CODE BLOCK
7. GraphQueries: Query NEO4J Graph with Python functions.
    # CODE BLOCK
```

Please note, that for functions 1. - 5. above, you also need to uncomment the following line:

```
 # onto_file_path_or_url: str = path_ontos.as_posix() + "/onto4/Ontology4.ttl"
```

B. Parameters

Most of the parameters to be passed to these functions are Python "Enums". For instance, for the function ...

execute_graph_queries()

```
""" 7. GraphQueries: Query NEO4J Graph with Python functions. """

execute_graph_queries(esrs_1=ESRS.EmissionsToAirByPollutant,
                      company=Company.Adidas,
                      periods=['2023', '2022'],
                      return_df=True,
                      stat=Stats.SUM,
                      esrs_2=ESRS.NetRevenue,
                      comp_prop=CompProp.Industries,
                      print_queries=False)
```

... the parameters are the Enums "ESRS", "Company", "Stats" and "CompProp".

These Enums allow you to easily select a value from the possible values such as "Adidas" after typing "Company." as your IDE should now show you all the possible values.

These Enum values are:

ESRS:

The ESRS-values refer to the 21 exemplary ESRS data points that you populated the KG with if you have (at least) run the functions 1. through 5. from "main.py". Please refer to the README-data.md-file in "/src/data/" for further details:

```
    AbsoluteValueOfTotalGHGEmissionsReduction
    AssetsAtMaterialPhysicalRiskBeforeClimateChangeAdaptationActions
    AssetsAtMaterialTransitionRiskBeforeClimateMitigationActions
    EmissionsToAirByPollutant
    EmissionsToSoilByPollutant
    EmissionsToWaterByPolllutant
    FinancialResourcesAllocatedToActionPlanCapEx
    FinancialResourcesAllocatedToActionPlanOpEx
    GrossLocationBasedScope2GHGEmissions
    GrossMarketBasedScope2GHGEmissions
    GrossScope1GHGEmissions
    GrossScope3GHGEmissions
    NetRevenue
    NetRevenueUsedToCalculateGHGIntensity
    TotalAmountOfSubstancesOfConcernGenerated
    TotalEnergyConsumptionFromFossilSources
    TotalEnergyConsumptionFromNuclearSources
```

```
    TotalEnergyConsumptionFromRenewableSources
    TotalGHGEmissions
    TotalUseOfLandArea
    TotalWaterConsumption
```

Company:

The Company-values refer to the 3 exemplary companies "Adidas", "BASF" and "Puma" that you populated the KG with. Please refer to the README-data.md-file in "/src/data/" for further details:

```
    Adidas
    BASF
    Puma
```

Stats:

The Stats-values refer to 4 statistical functions that can be used to calculate aggregates:

```
    MIN
    MAX
    AVG
    SUM
```

CompProp:

The two CompProp-values refer to Node properties of the "Company" Node which come from external sources such as wikidata or dbpedia. Aggregates and single data points can be calculated according to these values. Please refer to the functions in "G_graph_queries.py"":

```
    Country
    Industries
```

Please note that the sample JSON-files loaded into the KG only contains data for the periods 2022 and 2023.

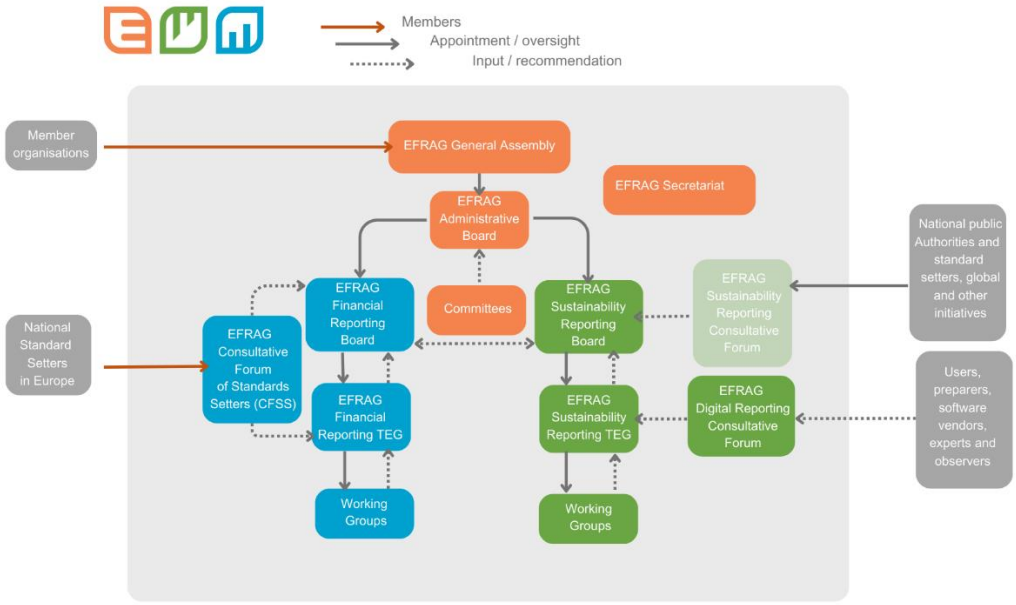
2. Research

This section explains some files in the "research"-folder.

I. ESRS

A. Drafts

ESRS stands for "European Sustainability Reporting Standards", a legislation coordinated by EFRAG. EFRAG is a private association serving the European Commission in Corporate reporting matters:



EFRAG outlined the ESRS requirements in multiple drafts covering the three sustainability areas "Environment", "Social" and "Corporate" governance ("ESG") that will take effect from 2024 on. As this project only focuses on the Environmental reporting requirements (the "E" in "ESG"), the "research" folder only contains the drafts that are concerned with environmental reporting requirements under ESRS:

```
- ESRS_GD: General Disclosures
- ESRS_GR: General Requirements
- ESRS_E1: Climate Change
- ESRS_E2: Pollution
- ESRS_E3: Water and marine resources
- ESRS_E4: Biodiversity and ecosystems
- ESRS_E5: Resource use and circular economy
```

The drafts define and describe the data points to be reported. The required data points were also aggregated in the Excel-table ESRS_Draft_10_2023.xlsx. In the "InNeo4j"-tab therein, we listed all the data points that were included in the sample JSON-files later to be loaded into the Knowledge Graph. We included at least one data point from each of the five draft sections (E1 to E5) (see the "Node" column in the individual worksheets).

Starting January 1, 2024, ESRS will apply to all companies whose size exceeds certain thresholds:

```
- Over 250 employees
```

```
- More than 40€ million in annual revenue
- More than 20€ million in total assets or balance sheet
- Publicly-listed equities who have more than 10 employees or 20€ million
  revenue
- International and non-EU companies with more than 150€ million annual
  revenue within the EU and which have at least one subsidiary or branch in
  the EU exceeding certain thresholds
```

Small-and-medium enterprises ("SMEs") are also required to report under a more limited set (ESRS LSME) (starting January 1, 2026) and apply to listed EU Member State companies who meet at least two of the three following criteria:

```
- Between 50-250 employees
- More than 8€ million in annual revenue (and less than 40€ million)
- More than 4€ million in total assets or balance sheet (and less than 20€
  million)
```

B. Reporting format

ESRS reports will have to be reported in a format that is machine-readable. EFRAG has stated that they will provide an ESRS digital taxonomy (a so-called XBRL taxonomy) that will form the basis of this electronic reporting. Please see also these drafts:

```
- ESRS_Implementation_Guidance.pdf
- ESRS_XBRL_Taxonomy.pdf
```

Companies today already publish their annual or quarterly reports in XBRL-format, but these XBRL-files do not yet contain ESG-data. From 2024 on, this will change as described above. Companies will probably aggregate existing corporate reporting with ESRS reporting in their annual reports and publish them both together in XBRL-format.

In this project the module "A_read_xbrl.py" was designed to read such XBRL-files. Please see the "README.md-file" in the root folder.

II. Ontologies

A. Introduction

The Knowledge Graph of this project is based on an Ontology developed in Protégé, a tool provided by Stanford University.

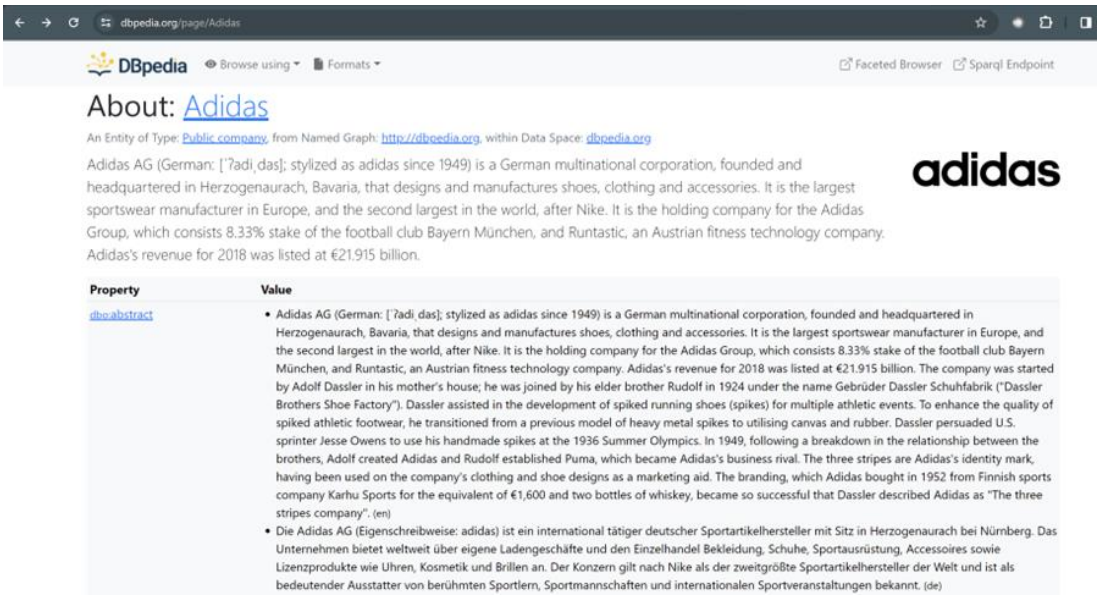
Ontologies are used to capture knowledge about some domain of interest. An ontology describes the concepts in the domain and also the relationships that hold between those concepts.

The most widely used ontology language is OWL from the World Wide Web Consortium (W3C) which stands for "Web Ontology Language". OWL is built on top of RDFS which itself is built on top of RDF (Resource Description Framework) to describe network graphs i.e., nodes and links.

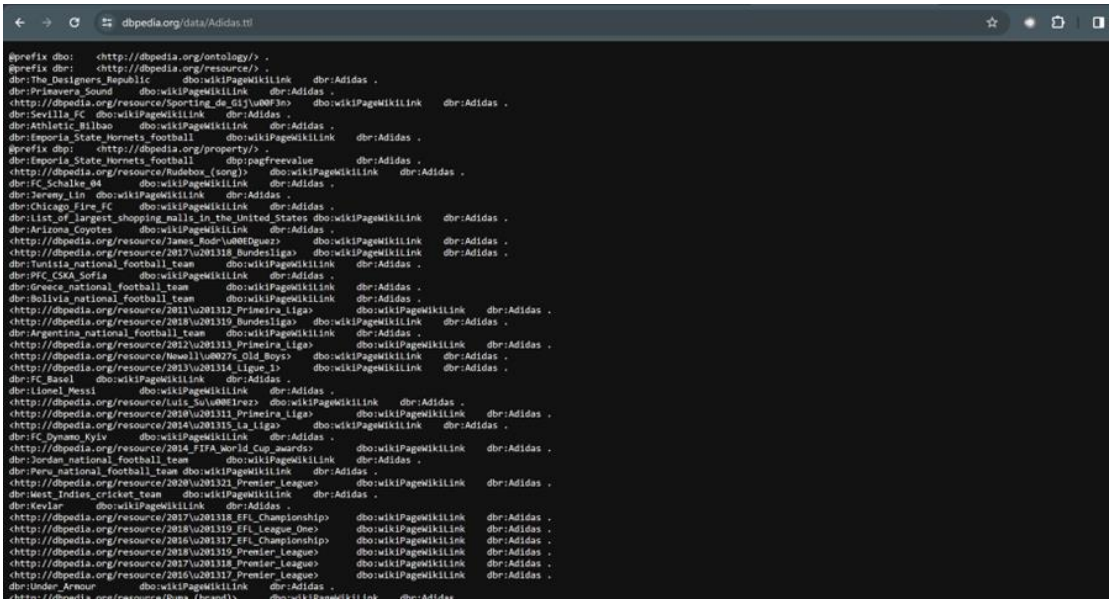
In OWL the fundamental construct is a triple consisting of a subject, predicate, and object. Triples can construct graphs because objects can be subjects and vice versa and so can be connected via predicate relationships. Such graphs are also known as triple stores.

Wikidata and dbpedia are public triple stores as they can be publicly accessed via http-requests. Every resource there can be represented as a collection of triples, which are both, machine-readable and human-readable.

For instance, the company "Adidas" in dbpedia can be represented in human-readable form ...



... or in machine-readable form ...



... if the format under "Formats" on top of the respective website is changed to a RDF-format such as Turtle (*.ttl-files). In the Turtle-representation of "Adidas", all the triples of subjects, predicates and objects are visible.

For instance, the fact that "Adidas" has EUR 6,364,000,000 in equity is represented by the triple:

```
dbr:Adidas    dbo:equity    "6.364E9"^^dbd:euro .
```

or explicitly:

```
SUBJECT: dbr:Adidas    PREDICATE: dbo:equity    OBJECT: "6.364E9"^^dbd:euro .
```

The prefixes "dbr:" and "dbo:" refer to previously defined namespaces that make definitions unique and unambiguous and differentiate between similar/equal terms in different concepts and domains.

B. Ontology research

The pdf-files under the "ontologies"-folder were used to learn about ontologies and how to construct them. We particularly recommend the hands-on "PizzaTutorial" (see folder "PizzaTutorial") for more information on the subject.

C. Ontology construction

As previously stated, the Knowledge Graph of this project is based on an Ontology developed in Protégé. After constructing the ontology there, it was saved as a file in Turtle format and in this project was named "Ontology4.ttl".

The Protégé tool distinguishes between "Classes", "Object Properties", "Data Properties" and "Individuals":

- **Classes:** Defines NODES such as the Node "Company"
- **Object Properties:** Defines graph RELATIONS between a DOMAIN (Source Node) and a RANGE (Target Node) such as "EMITS" between Source Node "Company" and Target Node "GHGEmission"
- **Data Properties:** Defines graph NODE PROPERTIES such as "Legal Entity Identfier" of a Node "Company"
- **Individuals:** Individuals are instances of Classes and contain real data. For instance, a Node "Company" with concrete Node properties of "LEI:"549300JSX0Z4CW0V5023"" and "label:Adidas"

The "Ontology4.ttl"-file contains "Classes", "Object Properties" and "Data Properties" but does not contain "Individuals" as the data later is imported separately. The "Ontology4.ttl"-file nevertheless contains the triples necessary to construct the Knowledge Graph schema:

D. "Emits" triples

For instance, the "Ontology4.ttl"-file contains the four triples to construct the "EMITS"-relationship:

```
###  http://www.semanticweb.org/rainergo/ontologies/2023/10/Ontology4#emits
rainergo:emits rdf:type owl:ObjectProperty ;
               rdfs:domain rainergo:Company ;
               rdfs:range rainergo:GHGEmission ,
                          rainergo:Scope1 .
```

These four triples are:

```
TRIPLE_1: subject: "emits"      predicate: "type"      object: "ObjectProperty"
TRIPLE_2: subject: "emits"      predicate: "domain"    object: "Company"
TRIPLE_3: subject: "emits"      predicate: "range"     object: "GHGEmission"
TRIPLE_4: subject: "emits"      predicate: "range"     object: "Scope1"
```

The namespaces rdf, owl and rdfs are publicly defined and accessible namespaces that are listed and defined in the first lines of a Turtle-file. The namespace "rainergo" is not publicly defined and accessible, but refers to the inherent namespace of this custom ontology.

E. Python libraries for RDFs

There are multiple Python libraries to construct and read such triple stores but the one that stands out in number of downloads and citations is rdflib. We used this library to read the triples from the "Ontology4.ttl"-file into a rdf graph and to construct cypher queries for the data import based on this ontology. Please see "B_read_graph.py".

F. SPARQL

SPARQL (*SPARQL Protocol And RDF Query Language*) is a query language that can read from and write to triple stores such as wikidata and dbpedia. It is comparable to SQL in relational databases. SPARQL queries in this project were used in "D_graph_construction.py" to populate the Knowledge Graph with external data from wikidata and dbpedia.

III. Knowledge Graphs

We collected some sources to learn about knowledge graphs, which are listed in the "LINKS"-file. A good resource for the most widely used knowledge graph software "NEO4J" and its proprietary query language Cypher can be found in the freely downloadable O'Reilly book Graph Databases.

3. Data

This section referes to the "data" folder in “src” which contains two sub-folders:

```
1) "XBRLs": XBRL-files of company's annual or quarterly reports. Currently, these
            files do NOT contain ESG-data.
2) "JSONs": JSON-files of company's sample ESG-data that later will be used to
            populate a NEO4J-Knowledge-Graph
```

I. XBRLs:

A. Where to get the files from?

Companies today already publish their annual or quarterly reports in XBRL-format, but these XBRL-files do not yet contain ESG-data. From 2024 on, this will change as ESRS requires companies whose size exceeds certain thresholds to publish ESG-data in the form of XBRL-files.

To show how such XBRL-files can be automatically parsed and read into JSON-files, here three sample XBRL-data-packages are provided for the following companies: Adidas AG ("adidasag.com"), BASF AG ("basfse.com") and Philips ("www.philips.com").

These XBRL-data-packages were downloaded from:

ESMA Databases and Registers - Corporate Reporting:

B. Files Format

The downloaded XBRL-data-packages must be organized in a certain manner so that the Python script "A_read_xbrl.py" later can parse and read these XBRLs into a JSON-file properly. The downloaded XBRL-files usually come in the form/hierarchy of:

```
- META-INF
- reports
- url-of-company
    - xbrl
        - report-year
            - file-name.xsd
            - file-name_cal.xml
            - file-name_def.xml
            - file-name_lab.xml
            - file-name_pre.xml
```

These files must be reorganized as follows:

```
- raw
    -> FOR ALL COMPANIES, COPY ALL "url-of-company"-FOLDERS (AND FILES THEREIN) INTO
       THIS "raw"-FOLDER:
    - url-of-company-1
    - url-of-company-2

    - reports
        -> FOR ALL COMPANIES, COPY ALL XHTML-FILES IN THE ORIGINAL "reports"-FOLDER
           INTO THIS "reports"-FOLDER:
        - file-name.xhtml

    -> LEAVE THE OTHER FOLDERS (SUCH AS "www.esma.europa.eu", "efrag.org",
       "www.xbrl.org", "xbrl.ifrs.org") UNTOUCHED AS THEY ARE NEEDED FOR PARSING THE
        FILES.
```

The XBRL-files now can be parsed and read into JSON-files with the Python script "A_read_xbrl.py".

If there are some problems while parsing, please refer to the attached document "XBRLs/XBRL_explained.pdf" or the GitHub page of the Python package that was used for parsing: py-xbrl

If an "xbrl.TaxonomyNotFound"-error occurs, please check the raised issue here: py-xbrl issues

II. JSONs:

As we wanted to inject some sample ESG data into the Knowledge-Graph, we created some exemplary JSON-files, assuming that such files in the course of the year 2024 will be extractable from the to-be-reported XBRL-files (see above). The required ESG data to be reported was outlined by ESMA and published in an Excel table: ESRS_Draft_10_2023.xlsx

For readability reasons and clarity, we did not want to inject sample data for all to-be-reported data points into the Knowledge-Graph, but only for some of them. We used the following 21 data points:

#	ParentNode	Node	Original Variable	Label	Unit	Unit Datatype
1	Energy	EnergyFromFossilSources	Total energy consumption from fossil sources	TotalEnergyConsumptionFromFossilSources	MWh	xsd:decimal
2	Energy	EnergyFromNuclearSources	Total energy consumption from nuclear sources	TotalEnergyConsumptionFromNuclearSources	MWh	xsd:decimal
3	Energy	EnergyFromRenewableSources	Total energy consumption from renewable sources	TotalEnergyConsumptionFromRenewableSources	MWh	xsd:decimal
4	Financials	Asset	Assets at material physical risk before considering climate change adaptation actions	AssetsAtMaterialPhysicalRiskBeforeClimateChangeAdaptationActions	EUR	xsd:decimal
5	Financials	Asset	Assets at material transition risk before considering climate mitigation actions	AssetsAtMaterialTransitionRiskBeforeClimateMitigationActions	EUR	xsd:decimal
6	Financials	Expenditure	Financial resources allocated to action plan (OpEx)	FinancialResourcesAllocatedToActionPlanOpEx	EUR	xsd:decimal
7	Financials	Expenditure	Financial resources allocated to action plan (CapEx)	FinancialResourcesAllocatedToActionPlanCapEx	EUR	xsd:decimal
8	Financials	Revenue	Net revenue used to calculate GHG intensity	NetRevenueUsedToCalculateGHGIntensity	EUR	xsd:decimal
9	Financials	Revenue	Net revenue	NetRevenue	EUR	xsd:decimal
10	GHGEmission	GHGEmission	Total GHG emissions	TotalGHGEmissions	tonsCO2Eq	xsd:decimal
11	GHGEmission	Reduction	Absolute value of total Greenhouse gas emissions reduction	AbsoluteValueOfTotalGHGEmissionsReduction	tonsCO2Eq	xsd:decimal
12	GHGEmission	Scope1	Gross Scope 1 greenhouse gas emissions	GrossScope1GHGEmissions	tonsCO2Eq	xsd:decimal
13	GHGEmission	Scope2	Gross location-based Scope 2 greenhouse gas emissions	GrossLocationBasedScope2GHGEmissions	tonsCO2Eq	xsd:decimal
14	GHGEmission	Scope2	Gross market-based Scope 2 greenhouse gas emissions	GrossMarketBasedScope2GHGEmissions	tonsCO2Eq	xsd:decimal
15	GHGEmission	Scope3	Gross Scope 3 greenhouse gas emissions	GrossScope3GHGEmissions	tonsCO2Eq	xsd:decimal
16	NaturalResource	Water	Total water consumption	TotalWaterConsumption	cubicmeter	xsd:decimal
17	NaturalResource	Land	Total use of land area	TotalUseOfLandArea	hectares	xsd:decimal
18	Pollution	Substance	Total amount of substances of concern that are generated or used during production or that are procured	TotalAmountOfSubstancesOfConcernGenerated	tons	xsd:decimal
19	Pollution	Waste	Emissions to air by pollutant	EmissionsToAirByPollutant	tons	xsd:decimal
20	Pollution	Waste	Emissions to soil by pollutant [+ by sectors/Geographical Area/Type of source/Site location]	EmissionsToSoilByPollutant	tons	xsd:decimal
21	Pollution	Waste	Emissions to water by pollllutant [+ by sectors/Geographical Area/Type of source/Site location]	EmissionsToWaterByPolllutant	tons	xsd:decimal

The "JSONs"-sub-folder contains the following exemplary JSON-files containing these data points for three companies ("Adidas", "BASF", "Puma") and two years (2022, 2023), altogether 6 JSON-files:

```
- Adidas_2022.json
- Adidas_2023.json
- BASF_2022.json
- BASF_2023.json
- Puma_2022.json
- Puma_2023.json
```

To create such exemplary json-files to be later read into the Knowledge-Graph, the template "templates/template.json" can be used.

4. Models

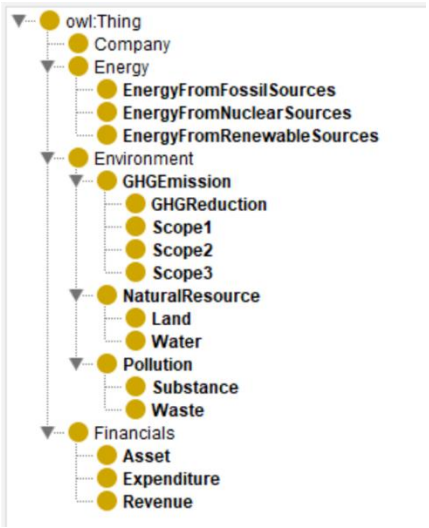
I. Files:

This models-folder contains Ontology-files and accompanying data. There is one ontology used in the project and thus only one ontology folder ("onto4"):

```
- onto4
    - Ontology4.ttl
    - params
    - data_needed
```

A. Ontology4.ttl

The name of the ontology ("Ontology4") has no special meaning and only reflects the latest version number. The ontology and file was created with the help of the free Protégé tool provided by Stanford University. The file format is Turtle (".ttl"). The ontology can be visually represented:



The ontology subordinates and hosts all 21 sample ESG data points referred to earlier:

#	ParentNode	Node	Original Variable	Label	Unit	Unit Datatype
1	Energy	EnergyFromFossilSources	Total energy consumption from fossil sources	TotalEnergyConsumptionFromFossilSources	MWh	xsd:decimal
2	Energy	EnergyFromNuclearSources	Total energy consumption from nuclear sources	TotalEnergyConsumptionFromNuclearSources	MWh	xsd:decimal
3	Energy	EnergyFromRenewableSources	Total energy consumption from renewable sources	TotalEnergyConsumptionFromRenewableSources	MWh	xsd:decimal
4	Financials	Asset	Assets at material physical risk before considering climate change adaptation actions	AssetsAtMaterialPhysicalRiskBeforeClimateChangeAdaptationActions	EUR	xsd:decimal
5	Financials	Asset	Assets at material transition risk before considering climate mitigation actions	AssetsAtMaterialTransitionRiskBeforeClimateMitigationActions	EUR	xsd:decimal
6	Financials	Expenditure	Financial resources allocated to action plan (OpEx)	FinancialResourcesAllocatedToActionPlanOpEx	EUR	xsd:decimal
7	Financials	Expenditure	Financial resources allocated to action plan (CapEx)	FinancialResourcesAllocatedToActionPlanCapEx	EUR	xsd:decimal
8	Financials	Revenue	Net revenue used to calculate GHG intensity	NetRevenueUsedToCalculateGHGIntensity	EUR	xsd:decimal
9	Financials	Revenue	Net revenue	NetRevenue	EUR	xsd:decimal
10	GHGEmission	GHGEmission	Total GHG emissions	TotalGHGEmissions	tonsCO2Eq	xsd:decimal
11	GHGEmission	Reduction	Absolute value of total Greenhouse gas emissions reduction	AbsoluteValueOfTotalGHGEmissionsReduction	tonsCO2Eq	xsd:decimal
12	GHGEmission	Scope1	Gross Scope 1 greenhouse gas emissions	GrossScope1GHGEmissions	tonsCO2Eq	xsd:decimal
13	GHGEmission	Scope2	Gross location-based Scope 2 greenhouse gas emissions	GrossLocationBasedScope2GHGEmissions	tonsCO2Eq	xsd:decimal
14	GHGEmission	Scope2	Gross market-based Scope 2 greenhouse gas emissions	GrossMarketBasedScope2GHGEmissions	tonsCO2Eq	xsd:decimal
15	GHGEmission	Scope3	Gross Scope 3 greenhouse gas emissions	GrossScope3GHGEmissions	tonsCO2Eq	xsd:decimal
16	NaturalResource	Water	Total water consumption	TotalWaterConsumption	cubicmeter	xsd:decimal
17	NaturalResource	Land	Total use of land area	TotalUseOfLandArea	hectares	xsd:decimal
18	Pollution	Substance	Total amount of substances of concern that are generated or used during production or that are procured	TotalAmountOfSubstancesOfConcernGenerated	tons	xsd:decimal
19	Pollution	Waste	Emissions to air by pollutant	EmissionsToAirByPollutant	tons	xsd:decimal
20	Pollution	Waste	Emissions to soil by pollutant [+ by sectors/Geographical Area/Type of source/Site location]	EmissionsToSoilByPollutant	tons	xsd:decimal
21	Pollution	Waste	Emissions to water by pollllutant [+ by sectors/Geographical Area/Type of source/Site location]	EmissionsToWaterByPolllutant	tons	xsd:decimal

The intent in the construction of the ontology was to use general concept class and Node names. This would allow subordination of further data points under the existing Nodes.

For instance: For the category "Scope 2 Greenhouse Gas Emissions" there are roughly 10 additional "Scope2"-related data points (i.e."gross", "net", "location based", "market based", etc.) to be reported (according to the ESRS_Draft_10_2023.xlsx) and loaded into the Knowledge-Graph later on. But in our sample JSON-file, there are only two of them, "GrossLocationBasedScope2GHGEmissions" and "GrossMarketBasedScope2GHGEmissions".

But this does not pose a problem, as these further data points can all be subordinated under the Node "Scope2", just with different Node labels.

The provided Node names try to generally cover the entire spectrum of environmental data ("E" in "ESG") to be reported under ESRS. The scope of our project was limited to environmental data and excludes social and governance aspects ("SG" in "ESG").

B. params

For all Nodes in the ontology, the following parameter data needs to be provided:

```
- unique_node_keys
- node_value_props
```

- unique_node_keys

Every Node in the ontology has certain properties that describe the Node. To identify and distinguish data points in the ontology and later in the Knowledge-Graph, certain Node properties need to be determined as unique_node_keys. unique_node_keys are comparable to primary keys in relational databases and are those Node properties that must be unique and that unambiguously identify a Node. They must be provided for each Node as a Python dictionary in the form:

```
unique_node_keys = {"node_label":["node_property_name"]}.

Example: {"Company":["LEI"]}
```

The *unique_node_keys* for the provided "Ontology4" are ["period", "label"] for all Nodes except for the Node "Company" whose unique_node_keys is the "Legal Entity Identifier" ("LEI"):

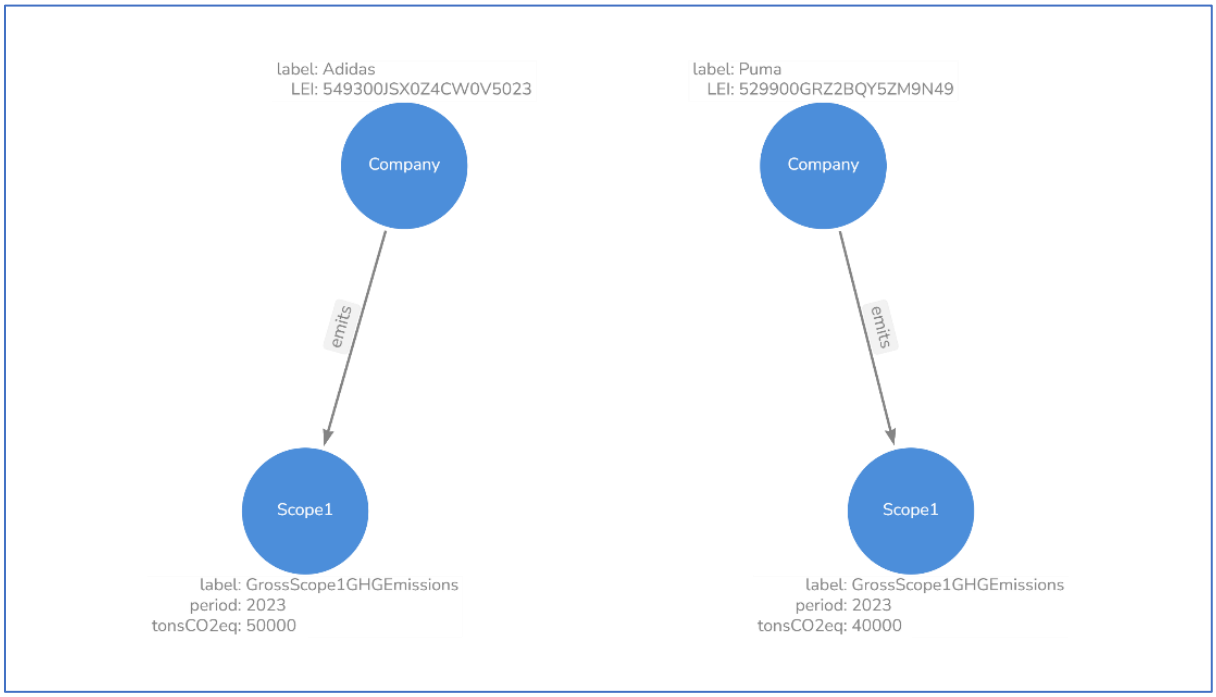
```
unique_node_keys = {
                    "Company": ["LEI"]
                    "Waste": ["period", "label"]
                    "Substance": ["period", "label"]
                    "EnergyFromNuclearSources": ["period", "label"]
                    "EnergyFromFossilSources": ["period", "label"]
                    "EnergyFromRenewableSources": ["period", "label"]
                    "GHGEmission": ["period", "label"]
                    "Scope1": ["period", "label"]
                    "Scope2": ["period", "label"]
                    "Scope3": ["period", "label"]
                    "GHGReduction": ["period", "label"]
                    "Land": ["period", "label"]
                    "Water": ["period", "label"]
                    "Asset": ["period", "label"]
                    "Expenditure": ["period", "label"]
                    "Revenue": ["period", "label"]
                    }
```

The "period" Node property usually refers to the business year of the respective company which mostly corresponds with the calendar year. The "label" property of a Node refers to its concrete data point name such as "GrossScope1GHGEmissions" for the Node "Scope1".

- node_value_props

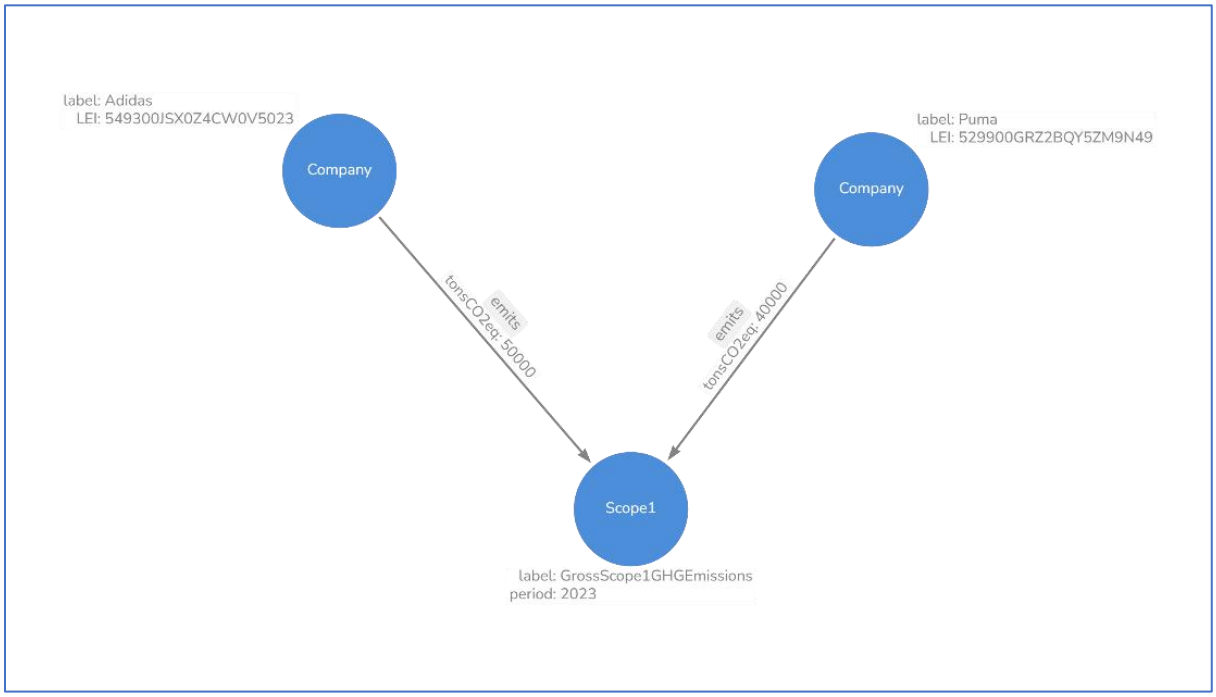
Node value properties ("node_value_props") are quantitative properties that are usually of datatype "decimal" as defined in the w3-consortium's XMLSchema ("http://www.w3.org/2001/XMLSchema#") and express the units a Node label (such as "Gross Scope 1 Greenhouse Gas Emissions") is measured in.

The Node value properties in the Knowledge-Graph could be stored in the Node itself, as depicted here:



The "Scope1"-Node on the left (picture above) with the label "GrossScope1GHGEmissions" for its "tonsCO2eq" property has a value of 50000. The "Company" with the label "Adidas" has an "emits"-relationship to this "Scope1"-Node indicating that "Adidas" emitted 50000 tons of CO2 equivalents "GrossScope1GHGEmissions" in 2023.

This structure has the disadvantage that each "Company"-Node needs one "Scope1"-Node to store the data. A better solution to this is to store the units "tonsCO2eq" in the relationship "emits" instead of the Node itself as depicted in the next picture:



For the "Scope1"-Node with the label "GrossScope1GHGEmissions" the property values are now stored in the incoming relationship "emits" for both "Companies", "Adidas" and "Puma".

With this approach, half of the target Nodes, i.e. Nodes with an incoming relationship, can be saved. All the Node properties provided in the "node_value_props"-dictionary will thus be stored in the incoming

relationship of the target Nodes. To later create queries in the Cypher query language, this has to be taken into consideration. As an example:

```
MATCH (s:SourceNode)-[r:Relationship]->[t:TargetNode]
```

If the TargetNode 't' has a quantity property 'EUR' to express the value of the TargetNode 't', then this property is the quantity property of the Relationship 'r'. In the example here, 'EUR' would be the quantity property of the Relationship 'r', if 'EUR' would be set in the node_value_props dictionary like so:

```
node_value_props = {'TargetNode':'EUR'}
```

For this reason, the node_value_props must also be provided as a Python dictionary. In the "Ontology4" the "node_value_props", i.e. the Node properties that are stored in incoming relationships, are:

```
node_value_props = {
                    "Waste": "tons",
                    "Substance": "tons",
                    "EnergyFromNuclearSources": "MWh",
                    "EnergyFromFossilSources": "MWh",
                    "EnergyFromRenewableSources": "MWh",
                    "GHGEmission": "tonsCO2Eq",
                    "Scope1": "tonsCO2Eq",
                    "Scope2": "tonsCO2Eq",
                    "Scope3": "tonsCO2Eq",
                    "GHGReduction": "tonsCO2Eq",
                    "Land": "hectares",
                    "Water": "cubicmetres",
                    "Asset": "EUR",
                    "Revenue": "EUR",
                    "Expenditure": "EUR"
                    }
```

C. data_needed

After extracting ESG data from xbrl-files, the data therein is stored in standardized JSON-files as explained before in the README-data.md-file. Exemplary JSON-files are located in "/src/data/JSONs/" for "Adidas", "BASF" and "Puma".

To load this data into the NEO4J Knowledge-Graph, Cypher queries first need to be generated. These Cypher queries must match the ontology represented by the "Ontology4.ttl"-file and must take the Node's peculiarities such as the unique_node_keys and the node_value_props (as described above) into account.

The Python script "B_rdf_graph.py" does exactly that. When initializing an instance of class "RDFGraph" passing a path to an ontology-file, it parses this ontology file, resolves all (subject-predicate-object) triples in there and builds a graph out of it. Passing the unique_node_keys and the node_value_props to the RDFGraph-method "create_query_templates()" will create four types of Cypher query templates:

```
Query Templates for:
                    - NODES
                    - RELATIONSHIPS
                    - CONSTRAINTS
                    - NAMESPACES
```

These Query templates are later used to load the data into the Knowledge-Graph.

Apply the *create_json_files_for_data_needed()-method*:

The RDFGraph-method "create_json_files_for_data_needed()" generates JSON-files with patterns for all Nodes and Relationships. These patterns are used to build the "node_template"-list and "relationship_template"-list in the "get_data_dicts()"-method in "C_read_data.py". They are stored in the "data_needed"-folder. All JSON-files in this folder must be represented in the "node_template"- and "relationship_template" in the "get_data_dicts()"-method.

Example:

Nodes

In the "data_needed" folder, there is, among others, an "Asset.json"-file with the following data:

```
{"Asset": {"label": "<HERE_label_VALUE>", "period": "<HERE_period_VALUE>"}}
```

This Node information must also appear in the "node_template"-list of the "get_data_dicts()"-method in "C_read_data.py", because "Asset" is a Node.

There are two "Asset"-Node items in this "node_template"-list which are:

```
{"Asset": {"label":
"AssetsAtMaterialPhysicalRiskBeforeClimateChangeAdaptationActions", "period":
company['period']}},
{"Asset": {"label": "AssetsAtMaterialTransitionRiskBeforeClimateMitigationActions",
"period": company['period']}}
```

These two "Asset"-Nodes correspond to two of the 21 data points.

The "<HERE_label_VALUE>" placeholder has been replaced by the Node names/labels "AssetsAtMaterialPhysicalRiskBeforeClimateChangeAdaptationActions" and "AssetsAtMaterialTransitionRiskBeforeClimateMitigationActions".

The "<HERE_period_VALUE>" placeholder was replaced by a "company"-dictionary (in the "get_data_dicts()"-method) that refers to the deserialized JSON-files such as "Adidas_2022.json", "Adidas_2023.json", etc. depending on which data is loaded.

Relationships

In the "data_needed" folder, there is also, among others, a "Company_emits_Scope1.json"-file with the following data:

```
{"Company_emits_Scope1":
    {"source": {"Company": {"LEI": "<HERE_LEI_VALUE>"}},
     "target": {"Scope1": {"period": "<HERE_period_VALUE>", "label":
"<HERE_label_VALUE>", "tonsCO2Eq": "<HERE_tonsCO2Eq_VALUE>"}}}}
```

This Relationship information must also appear in the "relationship_template"-list of the "get_data_dicts()"-method in "C_read_data.py", because "Company_emits_Scope1" represents the relationship "emits" between Node "Company" and Node "Scope1":

There is one "Company_emits_Scope1"-relationship in the "relationship_template"-list which is:

```
{"Company_emits_Scope1":
    {"source": {"Company": {"LEI": company['LEI']}},
     "target": {"Scope1": {"period": company['period'], "label":
"GrossScope1GHGEmissions", "tonsCO2Eq": company["GrossScope1GHGEmissions"]}}}}
```

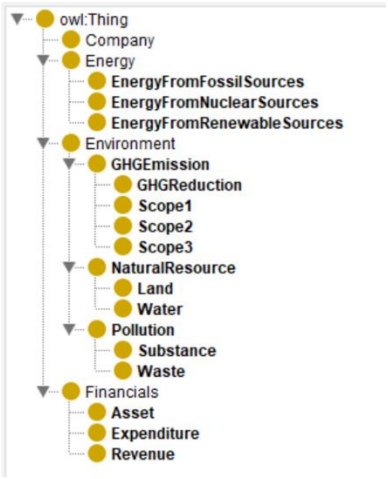
Again: The "<HERE_label_VALUE>" placeholder has been replaced by the Node name/label "GrossScope1GHGEmissions".

The "company"-dictionary that replaced the "<HERE_LEI_VALUE>"-, "<HERE_period_VALUE>"- and "<HERE_tonsCO2Eq_VALUE>"-placeholders refer to the deserialized JSON-files such as "Adidas_2022.json", "Adidas_2023.json", etc. depending on which data is loaded.

To repeat: All JSON-files in the "data_needed" folder must be represented in the "node_template"-list and "relationship_template"-list in the "get_data_dicts()"-method.

II. LOAD ADDITIONAL DATA:

Let's assume that not only 21 data points shall be loaded into the Knowledge-Graph, but far more than that. Here is the procedure of how to do that:




```
1. Extract the XBRL-data into JSON-files as described in the README-data.md file.
2. Check if all data points can be subordinated under the currently existing Nodes
   (see "Ontology4.ttl"-structure above):
   IF YOU NEED TO ADD FURTHER NODES, THEN:
    A. Remodel the ontolgy and create a new Ontolgy.ttl-file.
    B. Adjust the "unique_node_keys" and "node_value_props" in "params.py"
    C. Execute the create_json_files_for_data_needed()-method to get new
       "data_needed"-json-files as explained above
    -> Continue with 3.A.
3. If the new data can be subordinated under the currently existing Nodes, then:
    A. Check if the desired data is in the extracted JSON-file ("Adidas_2022.json",
       for instance)
    B. Make sure this JSON-file is stored in the folder "/src/data/JSONs/".
    C. Add the NODE data to the "node_template"-list of the "get_data_dicts()"-
       method as described above
        -> Make sure to replace the placeholder "<HERE_label_VALUE>" with the Node
           name/label of the new data, such as
           "MyNewScope1GHGEmissionsDataPointName"
    D. Add the RELATIONSHIP data to the "relationship_template"-list of the
       "get_data_dicts()"-method as described above
        -> Make sure to replace the placeholder "<HERE_label_VALUE>" with the Node
        name/label of the new data, such as "MyNewScope1GHGEmissionsDataPointName"
```

The new data can now be loaded into the Knowledge-Graph. Please see the README.md-file in the root folder of this project.

5. Modules

This section explains the logic and some implementation details of the individual modules in the “src”-folder.

I. A_read_xbrl.py:

The module is used for converting data in XBRL-files into standardized JSON-files that later can be used to populate a Knowledge Graph. The parameter for the only method "read_xbrl_to_json()" of the class XBRL is a Python Enum named "XHTMLName". This enum contains the names of the XHTML-files in the "reports"-folder, where all XHTML-files of XBRL-packages must be stored. Please refer to the README-data.md-file in the "/src/data/" directory. If reports are added there, their file names must also be added in this "XHTMLName"-Enum.

II. B_rdf_graph.py:

The module primarily uses the Python rdflib library. The class RDFGraph and its method fulfill three primary functions:

```
1) read the triples from the ontology-file into a graph:
            Method -> "rdf_graph.parse()" at instantiation
2) create JSON-files for the "data_needed"-folder:
            Method -> "create_json_files_for_data_needed()"
3) create cypher "data import"-query-templates:
            Method -> "create_query_templates()"
```

The class at instantiation needs a path to an ontology-file in Turtle-format, in our case the "Ontology4.ttl"-file. In addition, the methods also need the following Python dictionaries which, for the provided ontology, are currently located in "/src/models/Ontologies/onto4/params.py":

```
- unique_node_keys
- node_value_props
```

Please refer to the README-models.md-file for further details.

The results of the relevant methods of the class are printed if the module's main method is executed.

In these query templates, there are lines such as:

```
- WITH $node_data AS node_data ...
- WITH $rel_data AS rel_data ...
```

There, the "node_data" or "rel_data" refer to the dictionaries created in the "get_data_dicts()"-method of the module "C_read_data.py". The Cypher command "WITH" loads this dictionary data into memory.

III. C_read_data.py:

The module's only method "get_data_dicts()" reads the JSON-files in "src/data/JSONs/" into Python dictionaries. These dictionaries are passed as parameters to the "load_data_into_knowledge_graph()"-method of the module "D_graph_construction.py" to populate the knowledge graph. Please refer to the "README-models.md"-file.

IV. D_graph_construction.py:

The module has two primary responsibilities:

```
1) construct the NEO4j Knowledge Graph ("KG"):
    Methods:
        -> init_graph()
        -> load_onto_or_rdf()
2) load data into the NEO4J Knowledge Graph ("KG"):
    Methods:
        -> load_data_into_knowledge_graph()
        -> import_data_from_wikidata()
            -> Please note: In order to import the data, the method
               "import_wikidata_id()" must be run before.
        -> import_data_from_dbpedia()
        -> create_text_embedding()
```

init_graph()

This method sets all basic KG configurations. Please refer to the NEO4J neosemantics documentation for ontology-related settings.

load_onto_or_rdf()

This method imports the provided ontology into a NEO4J KG and creates the KG schema.

After executing both methods, "init_graph()" and "load_onto_or_rdf()", the KG schema (without data) can be displayed in the browser with the url "localhost:7474": NEO4J in browser

load_data_into_knowledge_graph()

This method loads the data from the JSON-files in "/src/data/JSONs/" into the KG as previously discussed and also laid out in the README-data.md and README-models.md-files.

import_data_from_wikidata()

This method imports external data from wikidata into the KG via SPARQL queries.

In most cases, only some of the seven parameters need to be provided:

```
Default parameters that most often are left untouched:
    - node_label:           The label of the Node where properties shall be stored.
                            Mostly this is "Company"
    - prop_name:            The Node property in wikidata. For Node "Company" this
                            is "LEI"
    - prop_wiki_id:         The wikidata-id for the above Node property. For "LEI",
                            this is "P1278"
Parameters that must be set:
    - new_prop_name:        The name of the new property from wikidata such as
                            "country" or "ISIN"
    - new_prop_wiki_id:     The wikidata-id for the new data such as "P17" for
                            "country" or "P946" for "ISIN"
    - use_new_prop_label:   Bool if the new wikidata data or its label shall be
                            used. For "country" the label would be the name of the
                            country (i.e. "Germany"), the data itself is another
                            wikidata-id such as "Q183" (for the country "Germany")
    - new_prop_is_list:     Bool if new wikidata data are multiple items or just one
                            item
```

import_data_from_dbpedia()

This method imports external data from dbpedia into the KG via SPARQL queries.

Here too, most often only some of the six parameters need to be provided:

```
Default parameters that most often are left untouched:
    - node_label:       The label of the Node where properties shall be stored.
                        Mostly this is "Company"
    - prop_name:        This is always the name of the class GraphConstruction
                        instance property "label_wikidata_id" (currently set to:
                        "wikidataID") as this links wikidata and dbpedia
Parameters that must be set:
    - prop_dbp_id:      The predicate that connects the "wikidataID" to the new
                        dbpedia "property" such as "owl:sameAs"
    - new_prop_name:    The name of the new property from dbpedia such as "abstract"
                        for the description of a company
    - new_prop_dbp_id:  The dbpedia-id for the new data such as "dbo:abstract" for
                        "abstract"
    - new_prop_is_list: Bool if new dbpedia data are multiple items or just one item
```

create_text_embedding()

This method enables to create text embeddings from LLMs for selected Node properties. Currently, the method creates a text embedding for the Node property "abstract" (i.e. a text description of the company) of Node "Company". This enables similarity search of Nodes in respect to their property "abstract". This method currently is not used actively as the similarity between company descriptions (i.e. "abstracts") does not make sense in our opinion. But it could make sense to do similarity search in the future if other text properties of the ESG data need to be compared.

The parameters of this method are:

```
- node_label:               The label of the Node for which a property shall be
                            embedded. Mostly this is "Company"
- node_primary_prop_name:   The primary and unique property name of the Node. This
                            is "LEI" for the Node "Company"
- prop_to_embed:            The name of the Node text property to be embedded. In
                            the example, this is "abstract", i.e. the description
                            of the company.
- vector_size:              Size of the embedding vector depending on the LLM to be
                            used. Currently for "bert-based-uncased", this is 768.
- similarity_method:        Similarity method to be used. Currently this is "cosine"
                            for "cosine similarity".
```

V. E_embeddings.py:

As outlined above, text properties of Nodes can be embedded. This module creates these text embeddings using the transformer library and the pretrained "bert-base-uncased"-model. Please refer to the documentation for further information.

VI. F_graph_bot.py:

This method uses the Python LangChain library and the OpenAI API to create a KG bot. In order to execute the functions, an OpenAI API KEY needs to be provided in the "secrets.env"-file. Please refer to the project's README.md-file.

The bot can be asked text questions which will be converted into cypher queries and sent to the KG. The query response from the KG is then again translated into a text message to be readable by humans.

The class GraphBot's "ask_question()"-method must be passed a question as string. It prints the question, the intermediate steps such as the created Cypher queries and the answer.

VII. G_graph_queries.py:

In this module, the class GraphQueries' methods create parameterized Cypher queries that can be used to answer a wide range of different questions. Some common parameters are:

```
esrs:       An ESRS-Enum, see project's README.md. Must always be provided.
periods:    Must be a list. Can be ['2022', '2023'] for multiple years or ['2022']
            for a single year.
company:    An Company-Enum. If set to "None", the result is shown for all comapnies
            in the KG.
stat:       A Stats-Enum ("Statistics"). If set to "None", indivdual values will be
            shown.
return_df:  If set to "True", a pandas "DataFrame" will be returned, else a NEO4J
            "Record"-object.
```

Some examples of possible questions to be answered by these queries can be found in the "main.py"-file under:

```
""" 7. GraphQueries: Query NEO4J Graph with Python functions """
```

6. Conclusion

In this documentation, we outlined the structure and the code of our project. We showed how to read XBRL-files into JSON-files later to be imported into a NEO4J Knowledge Graph. We created an ontology and Python code to translate this ontology into a graph structure and therefrom retrieve query templates. We used these query templates to import some exemplary data from standardized JSON-files. We additionally populated the graph with external data from wikidata and dbpedia. We wrote Python scripts to embed some text properties of Nodes and created a Knowledge Graph bot to answer human-readable questions. We also wrote flexible Python functions that can deliver answers to a wide range of sophisticated questions regarding the data in the Knowledge Graph.

We enjoyed working with this relatively new and exciting technology and hereby thank Prof. Simon for all his support and assistance.