EURITO Documentation

eurito

Nov 28, 2019

Contents:

Branch	Docs	Build
Master		
Development		

Welcome to EURITO! This repository houses our fully-audited tools and packages, as well as our in-house production system. If you're reading this on our GitHub repo, you will find complete documentation at our Read the Docs site.

CHAPTER 1

Data sources and core processing

In EURITO we predominantly use four data sources:

- EU-funded research from *CORDIS*
- Technical EU research on *arXiv*
- EU Patents from *PATSTAT*
- EU Companies [under license, we can't specify the source publicly. Contact us for more details!]

1.1 CORDIS

Data from the CORDIS's H2020 API and FP7 API funded projects is extracted using code found in this repository.

In total, 51250 organisations and 50640 projects were extracted from the API. There are 1102 proposal calls, 245465 publications and 34507 reports. In total 6545 are associated with the projects.

Software outputs are associated with the projects, using OpenAIRE API.

All of these entities are then linked together, and stored using a neo4j graph database. The code for automatically piping the data in neo4j is provided here.

1.1.1 Cordis to Neo4j

Tools for piping data from a SqlAlchemy ORM to Neo4j, to be used in the Luigi pipeline.

orm_to_neo4j(*session*, *transaction*, *orm_instance*, *parent_orm=None*, *rel_name=None*)
Pipe a SqlAlchemy ORM instance (a 'row' of data) to neo4j, inserting it as a node or relationship, as appropriate.

Parameters

- **session** (*sqlalchemy.Session*) – SQL DB session.
- **transaction** (*py2neo.Transaction*) – Neo4j transaction

- **orm_instance** (*sqlalchemy.Base*) – Instance of a SqlAlchemy ORM, i.e. a 'row' of data.
- **parent_orm** (*sqlalchemy.Base*) – Parent ORM to build relationship to
- **rel_name** (*str*) – Name of the relationship to be added to Neo4j

build_relationships(*session*, *graph*, *orm*, *data_row*, *rel_name*, *parent_orm=None*)

Build a py2neo.Relationship object from SqlAlchemy objects.x

Parameters

- **session** (*sqlalchemy.Session*) – SQL DB session.
- **transaction** (*py2neo.Transaction*) – Neo4j transaction
- **orm** (*sqlalchemy.Base*) – A SqlAlchemy ORM
- **rel_name** (*str*) – Name of the relationship to be added to Neo4j
- **parent_orm** (*sqlalchemy.Base*) – Another ORM to build relationship to. If this is not specified, it implies that `orm` is node, rather than a relationship.

Returns Relationships pointing to the node (inferred from ORM), and one pointing back to it's associated project.

Return type {relationship, back_relationship}

set_constraints(*orm*, *graph_schema*)

Set constraints in the neo4j graph schema.

Parameters

- **orm** (*sqlalchemy.Base*) – A SqlAlchemy ORM
- **graph_schema** (*py2neo.Graph.Schema*) – Neo4j graph schema.

prepare_base_entities(*table*)

Returns the objects required to generate a graph representation of the ORM.

Parameters **table** (*sqlalchemy.sql.Table*) – SQL alchemy table object from which to extract an graph representation.

Returns

Two ORMs and a string describing their relationship

Return type {orm, parent_orm, rel_name}

flatten(*orm_instance*)

Convert a SqlAlchemy ORM (i.e. a 'row' of data) to flat JSON.

Parameters **orm_instance** (*sqlalchemy.Base*) – Instance of a SqlAlchemy ORM, i.e. a 'row' of data.

Returns A flat row of data, inferred from *orm_instance*

Return type row (dict)

flatten_dict(*row, keys=[('title',), ('street', 'city', 'postalCode')]*)

Flatten a dict by concatenating string values of matching keys.

Parameters **row** (*dict*) – Data to be flattened

Returns Concatenated data.

Return type flat (str)

retrieve_node(*session*, *graph*, *orm*, *parent_orm*, *data_row*)
Retrieve an existing node from neo4j, by first retrieving it's id (field name AND value) via SqlAlchemy.

Parameters

- **session** (*sqlalchemy.Session*) – SQL DB session.
- **transaction** (*py2neo.Transaction*) – Neo4j transaction
- **orm** (*sqlalchemy.Base*) – SqlAlchemy ORM describing data_row
- **parent_orm** (*sqlalchemy.Base*) – Parent ORM to build relationship to
- **data_row** (*dict*) – Flat row of data retrieved from *orm*

Returns Node of data corresponding to data_row

Return type node (py2neo.Node)

table_from_fk(*fks*)
Get the table name of the fk constraint, ignoring the cordis_projects table

Parameters **fks** (list of SqlAlchemy.ForeignKey) – All foreign keys for a given table.

Returns The table name corresponding to the non-Project foreign key.

Return type tablename (str)

get_row(*session*, *parent_orm*, *orm*, *data_row*)
Retrieve a flat row of data corresponding to the parent relation, inferred via foreign keys.

Parameters

- **session** (*sqlalchemy.Session*) – SQL DB session.
- **parent_orm** (*sqlalchemy.Base*) – Parent ORM to build relationship to
- **orm** (*sqlalchemy.Base*) – SqlAlchemy ORM describing data_row
- **data_row** (*dict*) – Flat row of data retrieved from *orm*

Returns Flat row of data retrieved from *parent_orm*

Return type _row (dict)

1.1.2 Enrich Cordis with OpenAIRE

Tools for collecting OpenAIRE data (by Cordis project), and piping to Neo4j.

write_record_to_neo(*record*, *output_type*, *graph*)
A utility function, which takes record and writes it to neo4j graph

Parameters

- **record** (*dict*) – a dictionary that contains metadata about a record
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **graph** (*graph_session*) – connection to neo4j database

get_project_soups(*currentUrl*, *reqsession*, *output_type*, *projectID*)
Gets a beautiful soup according to output type and projectID

Parameters

- **currentUrl** (*str*) – URL to OpenAIRE API

- **reqsession** (*instance of Requests session*) – currently open HTTP request
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **projectID** (*str*) – EC project identifier

Returns a list of BeautifulSoup objects that contain the results from API call

Return type souplist(list)

get_results_from_soups (*souplist*)
Extracts string from all BeautifulSoup objects and merges them into one list

Parameters **souplist** (*list*) – a list of BeautifulSoup objects that contain the results from API call

Returns a list of strings with results metadata

Return type resultlist(list)

1.2 arXiv

All articles from arXiv, which is the world's premier repository of pre-prints of articles in the physical, quantitative and computational sciences, are already automatically collected, geocoded (using GRID) and enriched with topics (using MAG). Articles are assigned to EU NUTS regions (at all levels) using nesta's nuts-finder python package.

Data is transferred to EURITO's elasticsearch server via nesta's es2es package. The lolvelty algorithm is then applied to the data in order to generate a novelty metric for each article. This procedure is bettter described in this blog (see "Defining novelty").

The indicators using this data source are presented in this other EURITO repository.

In total, 1598033 articles have been processed, of which 459371 have authors based in EU nations.

1.3 PATSTAT

All patents from the PATSTAT service have been collected in nesta's own database using nesta's pypatstat library. Since this database is very large, we have selected patents which belong to a patent family with a granted patent first published after the year 2000, with at least one person or organisation (inventor or applicant) based in an EU member state. This leads to 1552303 patents in the database.

Data is transferred to EURITO's elasticsearch server via nesta's es2es package. The lolvelty algorithm is then applied to the data in order to generate a novelty metric for each article. This procedure is bettter described in this blog (see "Defining novelty").

The indicators using this data source are presented in this other EURITO repository.

1.4 Companies

We have acquired private-sector company data under license. The dataset contains 550540 companies, of which 133641 are based in the EU.

Data is transferred to EURITO's elasticsearch server via nesta's es2es package. The lolvelty algorithm is then applied to the data in order to generate a novelty metric for each article. This procedure is bettter described in this blog (see "Defining novelty").

The indicators using this data source are presented in this other EURITO repository.

CHAPTER 2

Batchables

2.1 run.py (lolvelty)

Calculates the "lolvelty" novelty score to documents in Elasticsearch, on a document-by-document basis. Note that this is a slow procedure, and the bounds of document "lolvelty" can't be known a priori.

run()

CHAPTER 3

Production pipelines

We use luigi routines to orchestrate our pipelines. The batching procedure relies on batchables as described in `batchables`. Other than `luigihacks.autobatch`, which is respectively documented in Nesta's codebase, the routine procedure follows the Luigi documentation well.

3.1 Transfer of Elasticsearch data

This pipeline is responsible for the transfer of Elasticsearch data from a remote origin (in our case, Nesta's Elasticsearch endpoint) to EURITO's endpoint.

class Es2EsTask(**args*, ***kwargs*)

Bases: `luigi.task.Task`

date = <luigi.parameter.DateParameter object>

origin_endpoint = <luigi.parameter.Parameter object>

origin_index = <luigi.parameter.Parameter object>

dest_endpoint = <luigi.parameter.Parameter object>

dest_index = <luigi.parameter.Parameter object>

test = <luigi.parameter.BoolParameter object>

chunksize = <luigi.parameter.IntParameter object>

do_transfer_index = <luigi.parameter.BoolParameter object>

db_config_path = <luigi.parameter.Parameter object>

output()

Points to the output database engine

run()

The task run method, to be overridden in a subclass.

See Task.run

class EsLolveltyTask(**args*, ***kwargs*)
Bases: nesta.core.luigihacks.estask.LazyElasticsearchTask

date = <luigi.parameter.DateParameter object>

origin_endpoint = <luigi.parameter.Parameter object>

origin_index = <luigi.parameter.Parameter object>

test = <luigi.parameter.BoolParameter object>

process_batch_size = <luigi.parameter.IntParameter object>

do_transfer_index = <luigi.parameter.BoolParameter object>

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

class RootTask(**args*, ***kwargs*)
Bases: luigi.task.WrapperTask

production = <luigi.parameter.BoolParameter object>

date = <luigi.parameter.DateParameter object>

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

3.2 Centrality Pipeline

Takes network from Neo4j database, calculates network centrality measures and updates each node in the database with new centrality attributes

class RootTask(**args*, ***kwargs*)
Bases: luigi.task.WrapperTask

The root task, which collects the supplied parameters and calls the main task.

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **production** (*bool*) – test mode or production mode

date = <luigi.parameter.DateParameter object>

output_type = <luigi.parameter.Parameter object>

production = <luigi.parameter.BoolParameter object>

requires()
Call the task to run before this in the pipeline.

class CalcCentralityTask(**args*, ***kwargs*)
Bases: luigi.task.Task

Takes network from Neo4j database, calculates network centrality measures and updates each node in the database with new centrality attributes

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **test** (*bool*) – run a shorter version of the task if in test mode

date = <luigi.parameter.DateParameter object>

output_type = <luigi.parameter.Parameter object>

test = <luigi.parameter.BoolParameter object>

output()
Points to the output database engine where the task is marked as done. The luigi_table_updates table exists in test and production databases.

run()
The task run method, to be overridden in a subclass.

See Task.run

3.3 Cordis to Neo4j

Task for piping Cordis data from SQL to Neo4j.

class CordisNeo4jTask(**args*, ***kwargs*)
Bases: luigi.task.Task

Task for piping Cordis data to neo4j

test = <luigi.parameter.BoolParameter object>

date = <luigi.parameter.DateParameter object>

output()
Points to the output database engine where the task is marked as done. The luigi_table_updates table exists in test and production databases.

run()
The task run method, to be overridden in a subclass.

See Task.run

class RootTask(**args*, ***kwargs*)
Bases: luigi.task.WrapperTask

production = <luigi.parameter.BoolParameter object>

requires()
The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a subclass can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See Task.requires

3.4 OpenAIRE to Neo4j

Pipe data directly from the OpenAIRE API to Neo4j by matching to Cordis projects already in Neo4j.

class RootTask(**args*, ***kwargs*)
Bases: `luigi.task.WrapperTask`

The root task, which collects the supplied parameters and calls the SimpleTask.

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **production** (*bool*) – test mode or production mode

date = <luigi.parameter.DateParameter object>

output_type = <luigi.parameter.Parameter object>

production = <luigi.parameter.BoolParameter object>

requires()
Call the task to run before this in the pipeline.

class OpenAireToNeo4jTask(**args*, ***kwargs*)
Bases: `luigi.task.Task`

Takes OpenAIRE entities from MySQL database and writes them into Neo4j database

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **output_type** (*str*) – type of record to be extracted from OpenAIRE API. Accepts "software", "datasets", "publications", "ECProjects"
- **test** (*bool*) – run a shorter version of the task if in test mode

date = <luigi.parameter.DateParameter object>

output_type = <luigi.parameter.Parameter object>

test = <luigi.parameter.BoolParameter object>

output()
Points to the output database engine where the task is marked as done. The luigi_table_updates table exists in test and production databases.

run()
The task run method, to be overridden in a subclass.

See Task.run

CHAPTER 4

Ontologies and schemas

4.1 Tier 0

Raw data collections ("tier 0") in the production system do not adhere to a fixed schema or ontology, but instead have a schema which is very close to the raw data. Modifications to field names tend to be quite basic, such as lowercase and removal of whitespace in favour of a single underscore.

4.2 Tier 1

Processed data ("tier 1") is intended for public consumption, using a common ontology. The convention we use is as follows:

- Field names are composed of up to three terms: a `firstName`, `middleName` and `lastName`
- Each term (e.g. `firstName`) is written in lowerCamelCase.
- `firstName` terms correspond to a restricted set of basic quantities.
- `middleName` terms correspond to a restricted set of modifiers (e.g. adjectives) which add nuance to the `firstName` term. Note, the special `middleName` term `of` is reserved as the default value in case no `middleName` is specified.
- `lastName` terms correspond to a restricted set of entity types.

Valid examples are `date_start_project` and `title_of_project`.

Tier 0 fields are implictly excluded from tier 1 if they are missing from the `schema_transformation` file. Tier 1 schema field names are applied via *nesta.packages.decorator.schema_transform*

CHAPTER 5

Scripts

A set of helper scripts for the batching system.

Note that this directory is required to sit in *$PATH*. By convention, all executables in this directory start with *nesta_* so that our developers know where to find them.

5.1 nesta_prepare_batch

Collect a batchable `run.py` file, including dependencies and an automaticlly generated requirements file; which is all zipped up and sent to AWS S3 for batching. This script is executed automatically in `luigihacks.autobatch.AutoBatchTask.run`.

Parameters:

- **BATCHABLE_DIRECTORY**: The path to the directory containing the batchable `run.py` file.
- **ARGS**: Space-separated-list of files or directories to include in the zip file, for example imports.

5.2 nesta_docker_build

Build a docker environment and register it with the AWS ECS container repository.

Parameters:

- **DOCKER_RECIPE**: A docker recipe. See `docker_recipes/` for a good idea of how to build a new environment.

CHAPTER 6

License

Python Module Index

e

Index

P

R

S

T

W