Knowledge Graph Embedding Server Documentation

Release 1.0-beta

Víctor Fernández Rico

Mar 26, 2018

Contents

CHAPTER 1

Introduction

You can skip this introduction if you know all about Knowledge graphs or you don't want to learn about them. Just go down to check how to install or run the service.

1.1 What are Knowledge Graphs?

There are many knowledge databases nowadays, and they are growing very quickly. Some of them are open and have a very broad domain, like [DBpedia](http://es.dbpedia.org/) or [Wikidata](http://wikidata.org/), both based in existent data on Wikipedia. Other knowledge databases are based on very specific domains, like datos.bne.es, which stores the information from Spanish National Library (*Biblioteca Nacional de España*) in an open, machine readable, way.

Most of those knowledge databases can be seen as **knowledge graphs**, where facts can be seen as triples: *head*, *label* and *tail*. This information is usually stored using semantic web tools, like RDF and can be queried through some languages like SPARQL.

1.2 What are Embeddings?

Embeddings are a way to represent all the relationships that exists on graphs, and they are commonly represented as multidimensional arrays. Those are useful to perform some machine learning tasks such as look for similar entities. With some embeddings models you can also do some simple algebraic operations with those arrays like adding them or substract and predict new entities.

1.3 What is this server?

This server provides a vertical solution on the machine learning area, going from the creation of datasets wich represents those knowledge graphs, to methods to perform queries such as look for similar entities given another. In the middle, the server provides training and indexing models that allows the query operations shown above.

1.4 What is included here?

The *vertical solution* depicted above is available as a Python library, so you can do a `python3 setup.py install` and that's all. But you can also deploy a web service using **docker** that is able to do almost every of those operations through a HTTP client. You can take a look to the documentation and discover all the things you can do on the *Table of contents*.

CHAPTER 2

Installation/Execution

You can use the Python library as is, or you can start a server, and use all the endpoints available.

2.1 Library installation

This repository provides a setuptools `setup.py` file to install the library on your system. It is pretty easy. Simply make `sudo python3 setup.py install` and it will install the library. Maybe some extra dependencies are required to run into your system, if so, you can execute this to get them all installed:

```
conda install scikit-learn scipy cython
```

And if you are using normal python3:

```
pip3 install numpy scipy pandas sympy nose scikit-learn
```

But the recommended way to getthe REST service working is to execute into the docker environment. You only need to have installed `docker` and `docker-compose` in your system.

2.2 Service execution

To run the service, go to images folder, execute `docker-compose up` and you will have a server on the port `localhost:6789` ready to listen HTTP requests.

```
cd images/
docker-compose up -d
curl http://localhost:6789/datasets
```

After this you will have an HTTP REST server listening to the API. but if you want to run the python library alone, you can connect to any of the docker containers created:

```
docker exec -it images_web_1 /bin/bash
```

If you are experiencing troubles when executing the docker image, check your user UID and change the user UID in **all** `Dockerfiles` inside `images/` folder. Then rebuild the images with: `docker-compose build --no-cache`

See more instructions about deployment at the *Server deployment* section.

2.3 Supported environment

The whole project has been built using Python 3.5 distributed by Anaconda, inside a docker image. If you want to run the development environment, just use this image `recognai/jupyter-scipy-kge`.

CHAPTER 3

Indices and tables

3.1 Table of contents

3.1.1 Modules

Dataset module

Introduction

The dataset class is used to create a valid object with the main idea to be used within the Experiment class. This approach allow us to create easily different models to be trained, leaving the complexity to the class itself.

You have different types of datasets. The main class of datasets is *kgeserver.dataset.Dataset*, but this will allow to create basic datasets from a csv or JSON file, without any restriction of triples or relations that may not be useful for the dataset and makes the binary file very big.

You have several Datasets that work with some of the most well known and free knowledge graphs on Internet. Those are:

- *WikidataDataset*: This class can manage all queries from the Wikidata portal, including wikidata id's prepended with Q, like in `Q1492`. It is fully ready to perform any you need with really good results.
- *ESDBpediaDataset*: This class is not as well ready as Wikidata, but is able to perform SPARQL Queries to get entities and relations on the Spanish DBpedia.

The most interesting feature that those Dataset class provides is to build a local dataset with making multiple parallel queries to the SPARQL endpoints to retrieve all information about a given topic. You can start by getting a *seed_vector* of the entities you want to focus on and then build a n levels graph by quering each entity to its relations with other new entities.

The seed vector can be obtained through the *load_from_graph_pattern* method. After that, you should save it on a variable and pass it as an argument to the *load_dataset_recurrently* method. This is the function that will make several queries to fill the dataset with the desried levels of depth.

To save the dataset into a binary format, you should use the *save_to_binary* method. This will allow to *open* the dataset later without executing any query.

Binary Dataset

The binary file of datasets are created using Pickle. It basicaly stores all the entities, all the relations and all the triples. It also stores some extra information to be able to *rebuild* the dataset later. The binary file is stored like a python dictionary which contains the following keys: `__class__`, `relations`, `test_subs`, `valid_subs`, `train_subs` and `entities`.

The `relations` and `entities` entries are lists, and it's length indicates us the number of relations or entities the dataset has. The `__class__` entry is for internal use of the class kgeserver.dataset. The triples are stored in three different entries, called `test_subs`, `valid_subs` and `train_subs`. Those subsets are created to be used for the next module, the algorithm module, wich will evaluate the dataset. This is a common practice when machine learning algorithms are used. If you need all the triples, they can be joined easily in python by adding the three lists between them:

```
triples = dataset["test_subs"] + dataset["valid_subs"] + dataset["train_subs"]
```

The split ratio commonly used is to use the 80% of the triples to train and the rest of triples are divided equally between *test* and *valid* triples. You can create a different split providing a value to *dataset.train_split*. It also exists an *dataset.improved_split* method wich takes a bit longer to create, but it is better to test the dataset.

Dataset Class

This class is used to create basic datasets. They can be filled with csv files, JSON files or even simple sparql queries.

Methods

Here is shown all the different methods to use with dataset class

WikidataDataset

This class will enable you to generate a dataset from the information present in Wikidata Knowledge base. This class only needs to get a simple graph pattern to get started to build a dataset. An example of graph pattern that should be passed to *WikidataDataset.load_from_graph_pattern* method:

```
"{ ?subject wdt:P950 ?bne . ?subject ?predicate ?object }"
```

It is required to bind at least three variables, because they will be used in the next queries. Those variables should be called "?subject", "?predicate" and "?object".

Methods

ESDBpediaDataset

In a similar way that it occurs with *WikidataDataset*, this class will allow to you to create datasets from the spanish DBpedia. The graph pattern you should pass to *ESDBpediaDataset.load_from_graph_pattern* method looks like this:

```
{ ?subject dcterms:subject <http://es.dbpedia.org/resource/Categoría:Trenes_de_alta_
↪velocidad> . ?subject ?predicate ?object" }
```

As for *WikidataDataset*, you need to bind the same three variables: "?subject", "?predicate" and "?object".

Methods

Algorithm module

This module contains several Class. The main purpose of the module is to provide a clear training interface. It will train several models with several distinct configs and choose the best one. After this, it will create a ModelTrainer class ready to train the entire model.

Methods

Here is shown all the different methods to use with dataset class

Experiment class

This class is a modified version of the file which can be found on https://github.com/mnick/holographic-embeddings/tree/master/kg/base.py, and was created by Maximilian Nickel mnick@mit.edu.

Methods

Here is shown all the different methods to use with experiment class

Server module

The server class is used to predict triples or statements, or find similar entities on the model.

Server class

Here is shown all the different methods to use with Server class

SearchIndex Class

This class is used to provide an extra layer to the server. Can perform loads and savings to disk of the index.

Here is shown all the different methods to use with SearchIndex class

3.1.2 REST Service

El servicio REST está compuesto principalmente de un recurso dataset con distintas operaciones

Endpoints

Aquí se detallarán todos los endpoints del servicio. El valor de la prioridad que se muestra indica la importancia que se le va a dar a la implementación de ese servicio. Cuanto menor sea, más importancia se le dará.

Datasets management

The */dataset* collection contains several methods to create, add triples to the dataset, train and generate search indexes.

It also contains these main params

```
{"entities", "relations", "triples", "status", "algorithm"}
```

The `algorithm` parameter contains all the information the dataset are trained with. See */algorithm* collection to get more information about this.

Dataset will be changing its status when actions such training or indexing are performed. The *status* can only grow up. When a changing status is taking place, the dataset cannot be edited. In this situations, the status will be a negative integer.

status: `untrained` -> `trained` -> `indexed`

GET /datasets/(**int:** *dataset_id*)**/**
Get all the information about a dataset, given a `dataset_id`

Sample request and response

`GET /datasets/1/`

```
{
        "dataset": {
                "relations": 655,
                "triples": 3307248,
                "algorithm": {
                        "id": 2,
                        "embedding_size": 100,
                        "max_epochs": null,
                        "margin": 2
                },
                "entities": 651759,
                "status": 2,
                "name": null,
                "id": 4
        }
}
```

Parameters

- **dataset_id** (*int*) – Unique *dataset_id*

Status Codes

- 200 OK – Returns all information about a dataset.
- 404 Not Found – The dataset can't be found.

POST /datasets/(**int:** *dataset_id*)**/train?algorithm=**
int: *id_algorithm* Train a dataset with a given algorithm id. The training process can be quite large, so this REST method uses a asynchronous model to perform each request.

The response of this method will only be a `202 ACCEPTED` status code, with the `Location:` header filled with the task path element. See `/tasks` collection to get more information about how tasks are managed on the service.

The dataset must be in a 'untrained' (0) state to get this operation done. Also, no operation such as `add_triples` must be being processed. Otherwise, a 409 CONFLICT status code will be obtained.

Parameters

- **dataset_id** (*int*) – Unique *dataset_id*

Query Parameters

- **id_algorithm** (*int*) – The wanted algorithm to train the dataset

Status Codes

- 202 Accepted – The requests has been accepted to the system and a task has been created. See Location header to get more information.
- 404 Not Found – The dataset or the algorithm can't be found.
- 409 Conflict – The dataset cannot be trained due to its status.

GET /datasets/

Gets all datasets available on the system.

Status Codes

- 200 OK – All the datasets are shown correctly

POST /datasets?dataset_type=(**int:** *dataset_type*)

Creates a new and empty dataset. To fill in you must use other requests.

You also must provide `dataset_type` query param. This method will create a WikidataDataset (id: 1) by default, but you also can create different datasets providing a different dataset_type.

Inside the body of the request you can provide a name and/or a description for the dataset. The name must be unique. For example:

Sample request

```
POST /datasets
```

```
{"name": "films", "description": "A dataset with favourite films"}
```

Sample response

The `location:` header of the response will contain the relative URI for the created dataset. Additionally, the body of the response will contain a dataset object with only id argument filled in:

```
location:  /datasets/32
```

```
{
    "dataset": {
        "id": 32
    }
}
```

Query Parameters

- **dataset_type** (*int*) – The dataset type to be created. 0 is for a simple Dataset and 1 is for WikidataDataset (default).

Status Codes

- 201 Created – A new dataset has been created successfuly. See `Location:` header to get the id and the new resource path.
- 409 Conflict – The given name already exists on the server.

PUT /datasets/(**int:** *dataset_id*)
Edits the description from a existing dataset.

Sample request

```
PUT /datasets
```

```
{"description": "A dataset with most awarded films"}
```

Parameters

- **dataset_id** (*int*) – Unique *dataset_id*

Status Codes

- 200 OK – The dataset has been updated successfully. The updated dataset will be returned in the response.
- 404 Not Found – The provided *dataset_id* does not exist.

POST /datasets/(**int:** *dataset_id*)**/triples**
Adds a triple or a list of triples to the dataset. You must provide a JSON object on the request body, as shown below on the example. The name of the JSON object must be *triples* and must contain a list of all entities to be introduced inside the dataset. These entities must contain `{"subject", "predicate", "object"}` params. This notation is similar to other known as *head*, *label* and *tail*.

Only triples can be added on a `untrained` (0) dataset.

Ejemplo

```
POST /datasets/6/triples
```

```
{"triples": [
        {
            "subject": {"value": "Q1492"},
            "predicate": {"value": "P17"},
            "object": {"value": "Q29"}
        },
        {
            "subject": {"value": "Q2807"},
            "predicate": {"value": "P17"},
            "object": {"value": "Q29"}
        }
    ]
}
```

Parameters

- **dataset_id** (*int*) – Unique *dataset_id*

Status Codes

- 200 OK – The request has been successful
- 404 Not Found – The dataset or the algorithm can't be found.
- 409 Conflict – The dataset cannot be trained due to its status.

POST /datasets/(**int:** *dataset_id*)**/generate_triples**

Adds triples to dataset doing a sequence of SPARQL queries by levels, starting with a seed vector. This operation is supported only by certain types of datasets (the default one, type=1)

The request will use asyncronous operations. This means that the request will not be satisfied on the same HTTP connection. Instead, the service will return a */task* resource that will be queried with the progress of the task.

The `graph_pattern` argument must be the where part of a SPARQL query. It **must** contain three variables named as `?subject`, `?predicate` and `?object`. The service will try to make a query with these names.

You also must provide the `levels` to make a deep lookup of the entities retrieved from previous queries.

The optional param `batch_size` is used on the first lookup for SPARQL query. For big queries you must tweak this parameter to avoid server errors as well as to increase performance. It is the LIMIT statement when doing this queries.

Sample request

```
{
    "generate_triples":
        {
            "graph_pattern": "SPARQL Query",
            "levels": 2,
            "batch_size": 30000
        }
}
```

Sample response

The `location:` header of the response will contain the relative URI for the created task resouce. Additionally, it is possible to get the task id from the response in json format.

```
location:  /tasks/32
```

```
{
    "message": "Task 32 created successfuly",
    "status": 202,
    "task": {
        "id": 32
    }
}
```

Parameters

- **dataset_id** (*int*) – Unique identifier of dataset

Status Codes

- 404 Not Found – The provided *dataset_id* does not exist.
- 409 Conflict – The *dataset_id* does not allow this operation
- 202 Accepted – A new task has been created. See /tasks resource to get more information.

POST /datasets/(**int:** *dataset_id*)**/embeddings**

Retrieve from the trained dataset the embeddings from a list of entities.

If on the request list the user requests for a entity that does not exist, the response won't contain that element. The 404 error is limited to the dataset, not the entities inside the dataset.

The dataset must be in trained status (>= 1), because a model must exist to extract triples from. If not, a 409 CONFLICT will be returned.

This could be useful if it is used with /similar_entities endpoint, to find similar entities given a different embedding vector.

Sample request

```
POST /datasets/6/embeddings
```

```
{"entities": [
    "http://www.wikidata.org/entity/Q1492",
    "http://www.wikidata.org/entity/Q2807",
    "http://www.wikidata.org/entity/Q1" ]
}
```

Sample response

```
{ "embeddings": [
    [
        "Q1",
        [0.321, -0.178, 0.195, 0.816]
    ],
    [
        "Q2807",
        [-0.192, 0.172, -0.124, 0.138]
    ],
    [
        "Q1492",
        [0.238, -0.941, 0.116, -0.518]
    ]
  ]
}
```

Note: The upper vectors are only shown as illustrative, they are not real values

Parameters

- **dataset_id** (*int*) – Unique id of the dataset

Status Codes

- 200 OK – Operation was successful
- 404 Not Found – The dataset ID does not exist
- 409 Conflict – The dataset is not on a correct status

POST /datasets/(int: *dataset_id*)/generate_index?n_trees=(int: *n_trees*) Generates an Spotify Annoy index to use dataset services. The execution of this action is needed to use **triples-prediction_** services.

See more info on Server module.

Sample request

```
POST /datasets/1/generate_index
```

Sample response

```
{
    "status": 202,
    "message": "Task 4 created successfuly"
}
```

Parameters

- **dataset_id** (*int*) – Unique id of the dataset
- **n_trees** (*int*) – Number of trees to generate with Annoy

Status Codes

- 202 Accepted – The request has been accepted in the system and a task has been created. See Location header to get more information.
- 404 Not Found – The dataset can't be found.
- 409 Conflict – The dataset cannot be trained due to it's status.

POST /datasets/(int: *dataset_id*)/generate_autocomplete_index

Creates a task to build an autocomplete index

The task will perform a request to SPARQL endpoint for each entity. This will extract the labels, description and altLabels and store it on an Elasticsearch database.

It is also possible give the languages desired to build the autocomplete index, allowing not only having english language, but others available on the endpoint. You must specify in the body a param named *langs* with a list with all language codes in ISO 639-1 format.

Sample request

```
POST /datasets/6/generate_autocomplete_index
```

```
{
    "langs" :  [
        "en", "es"
    ]
}
```

Sample response

```
{
    "status": 202,
    "message": "Task 73 created successfuly"
}
```

Parameters

- **dataset_id** (*int*) – Unique id of the dataset

Status Codes

- 202 Accepted – The request has been accepted in the system and a task has been created. See Location header to get more information.
- 404 Not Found – The dataset can't be found.
- 409 Conflict – The dataset cannot be trained due to it's status.

Algorithms

The algorithm collection is used mainly to create and see the different algorithms created on the server.

The hyperparameters that are allowed currently to tweak are: - *embedding_size*: The size of the embeddigs the trainer will use - `margin`: The margin used on the trainer - `max_epochs`: The maximum number of iterations of the algorithm

GET /algorithms/
Gets a list with all the algorithms created on the service.

GET /algorithms/(**int:** *algorithm_id*)
Gets only one algorithm

Parameters

- **algorithm_id** (*int*) – The algorithm unique identifier

POST /algorithms/
Create one algorithm on the service. On success, this method will return a 201 CREATED status code and the header parameter *Location:* filled with the relative path to the created resource.

The body of the request must contain all parameters for the new algorithm. See the example below:

Sample request

```
POST /algorithms
```

```
{
        "algorithm": {
                "embedding_size": 50,
                "margin": 2,
                "max_epochs": 80
        }
}
```

Sample successfull response The response when creating a new algorithm gives the location header filled with the URI of the new resource. It also returns the HTTP 202 status code, and the body has information about the request in json format.

```
location:  /algorithm/2
```

```
{
    "status": 202,
    "algorithm": {
        "id": 2
    },
    "message": "Algorithm 2 created successfuly"
}
```

Status Codes

- 201 Created – The request has been processed successfuly and a new resource has been created. See `Location:` header to get the new path.

Tasks

The task collection stores all the information that async request need. This collection are made mainly to get the actual state of tasks, but no to edit or delete tasks (Not implemented).

GET /tasks/(**int:** *task_id*) **?get_debug_info=boolean:** *get_debug_info***&?no_redirect=boolean:** *no_redirect* Shows the progress of a task with a `task_id`. The finished tasks can be deleted from the system without previous advise.

Some tasks can inform to the user about its progress. It is done through the progress param, which has *current* and *total* relative arguments, and *current_steps* and *total_steps* absolute arguments. When a task involves some steps and the number of small tasks to be done in next step cannot be discovered, the current and total will only

indicate progress in current step, and will not include previous step, expected to be already done, or next step which is expected to be empty.

The resource has two optional parameters: `get_debug_info` and `no_redirect`. The first one, `get_debug_info` set to true on the query params will return additional information from the task. The other param, `no_redirect` will avoid send a 303 status to the client to redirect to the created resource. Instead it will send a simple 200 status code, but with the location header filled.

Parameters

- **task_id** (*int*) – Unique *task_id* from the task.

Status Codes

- 200 OK – Shows the status from the current task.
- 303 See Other – The task has finished. See Location header to find the resource it has created/modified. With `no_redirect` query param set to true, the location header will be filled, but a 200 code will be returned instead.
- 404 Not Found – The given *task_id* does not exist.

Triples prediction

GET /datasets/(**int:** *dataset_id*)**/similar_entities/**
string: *entity***?limit=int:** *limit***?search_k=int:** *search_k*

POST /datasets/(**int:** *dataset_id*)**/similar_entities?limit=**
int: *limit***?search_k=int:** *search_k* Get the *limit* entities most similar to a *entity* inside a *dataset_id*. The given number in *limit* excludes the entity given itself.

The POST method allows any representation of the wanted resource. See the example below. You can provide an entity as an URI or other similar representation, even an embedding. The type param inside entity JSON object must be "uri" for a URI or similar representation and "embedding" for an embedding.

The `search_k` param is used to tweak the results of the search. When this value is greater, the precission of the results are also greater, but the time it takes to find the response is also bigger.

Sample request

```
GET /datasets/7/similar_entities?limit=1&search_k=10000
```

```
{ "entity":
      {"value": "http://www.wikidata.org/entity/Q1492", "type": "uri"}
}
```

Sample response

```
{    "similar_entities":
    {    "response":
        [
            {"distance": 0, "entity": "http://www.wikidata.org/entity/Q1492"},
            {"distance": 0.8224636912345886, "entity": "http://www.wikidata.org/
↪entity/Q15090"}
        ],
        "entity": "http://www.wikidata.org/entity/Q1492",
        "limit": 2
    },
    "dataset": {
        "entities": 664444,
```

```
        "relations": 647,
        "id": 1,
        "status": 2,
        "triples": 3261785,
        "algorithm": 100
    }
}
```

Parameters

- **dataset_id** (*int*) – Unique id of the dataset

Query Parameters

- **limit** (*int*) – Limit of similar entities requested. By default this is set to 10.
- **search_k** (*int*) – Max number of trees where the lookup is performed. This increase the result quality, but reduces the performance of the request. By default is set to -1

Status Codes

- 200 OK – The request has been performed successfully
- 404 Not Found – The dataset or the entity can't be found

POST /datasets/(int: *dataset_id*)/distance

Returns the distance between two elements. The lower the number is, most probable to be both the same triple. The minimum distance is 0.

Request Example

```
POST /datasets/0/similar_entities
```

```
{
    "distance": [
        "http://www.wikidata.org/entity/Q1492",
        "http://www.wikidata.org/entity/Q5682"
    ]
}
```

HTTP Response

```
{
    "distance": 1.460597038269043
}
```

Parameters

- **dataset_id** (*int*) – Unique id of the dataset

Status Codes

- 200 OK – The request has been performed successfully
- 404 Not Found – The dataset or the entity can't be found

POST /datasets/(int: *dataset_id*)/suggest_name

Gives a list of autocomplete suggestions. For each entity, this will show labels on every language available, descriptions and altLabels.

If any suggestion is available, this will return an empty list.

Request Example

`POST /datasets/7/suggest_name`

```
{
    "input": "human"
}
```

HTTP Response

```
[
    {
        "text": "humano",
        "entity": {
            "alt_label": {
                "es": [
                    "humano",
                    "Homo sapiens sapiens",
                    "persona",
                ],
                "en": [
                    "people",
                    "person",
                    "human being"
                ]
            },
            "label": {
                "en": "human",
                "es": "ser humano",
            },
            "entity": "Q5",
            "description": {
                "en": "common name of Homo sapiens (Q15978631), unique extant species of the genus Homo",
                "es": "especie animal perteneciente a la familia Hominidae, única superviviente del género Homo",
            }
        }
    }
]
```

Parameters

- **dataset_id** (*int*) – Unique id of the dataset

Status Codes

- 200 OK – The request has been performed successfully
- 404 Not Found – The dataset or the entity can't be found

3.1.3 Service Architecture

The service has an architecture based in docker containers. Currently it uses three different containers:

- **Web container**: This container exposes the only open port of all system. Provides a `gunicorn` web server that accepts HTTP requests to the REST API and responds to them.

- **Celery Container**: This container is running on the background waiting for a task on its queue. It contains all library code and `celery`.
- **Redis container**: The redis key-value storage is a dependency from Celery. It also stores all the progress of the tasks running on Celery queue.

Server deployment

The old version of this repository didn't had any Dockerfile or image available to run the code. This has changed, and two containers has been created to hold both web server and asyncronous task daemon (celery).

Also, a simple container orchestation with docker-compose has been used. You can see all the information inside images/ folder. It contains two Dockerfiles and a docker-compose.yml that allows to build instantly the two images and connect the containers. To run them you only have to clone the entire repository and execute those commands:

```
cd images/
docker-compose build
docker-compose up
```

The previous method is still available if you can't use docker-compose on your machine

Images used

The previous image used on developement environment was `recognai/jupyter-scipy-kge`. This image contains a lot of code that the library and rest service does not use.

Using `continuumio/miniconda3` docker image as base, it is possible to install only the required packages, minimizing the overall size of the container.

Both containers will launch a script on startup that will reinstall the kge-server package on python path, to get latest developement version running, and then will launch the service itself: gunicorn or celery worker.

Standalone containers to use in production are not still available.

Filesystem permissions

The images used creates a new user called *kgeserver* with 1001 as its UID and owns to the users group. This is helpful for a user running in development environment. But the docker-compose file mounts some folders from host machine that can create some `PermissionError` exceptions. To avoid them, always use write permissions for users group. You are also **free to modify the Dockerfile to solve the UID issues** you could have inside your system.

The `docker-compose` command will create inside both celery and web containers a data volume which is mounted on the root of the github repository.

Ports

Currently, while development is taking place, the port which is being used is **6789**, but you can change this easily on the `docker_compose.yml` file.

How to build documentation

This documentation page is built using sphinx framework, and is written in reStructuredText. To build some documentation strings, it needs to have some `python` libraries installed like `numpy` or `scikit-learn`. It exists one image to build this docs automatically, just running a docker container

```
cd images/sphinx-doc
docker build -t sphinx-doc .
docker run --rm -v $PWD/../..:/home/kgeserver/kge-server sphinx-doc html
```

You may noticed the last argument called `html`. It is the argument that will be passed to the make argument. In this case, calling to html will lead to a new `doc/build/html` folder with all new generated docs.

- genindex
- modindex
- search

HTTP Routing Table

/algorithms

/datasets

/datasets?dataset_type=(int:dataset_type)

/tasks