Clinical Knowledge Graph Documentation

Release 1.0

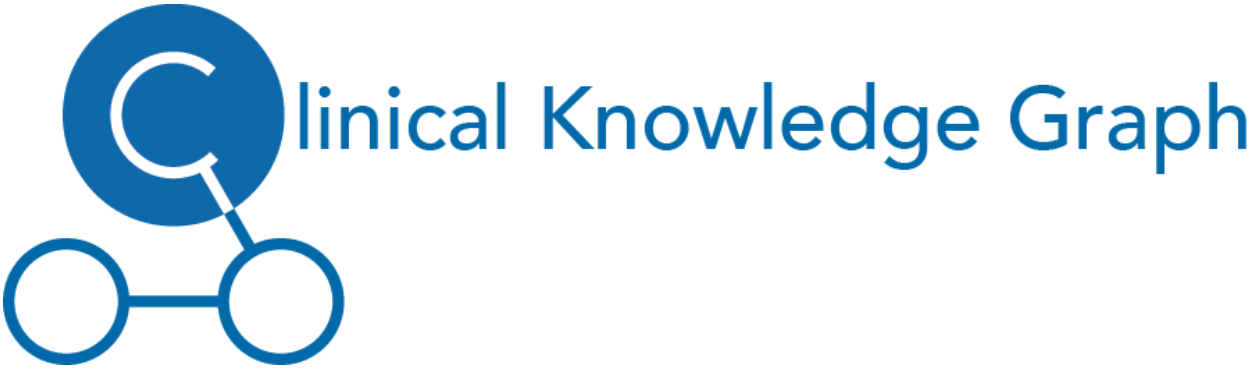
Alberto Santos, Ana Rita Colaço, Annelaura B. Nielsen

Sep 11, 2021

CONTENTS

This web page contains the documentation for the Clinical Knowledge Graph using **Sphinx**.

./ckg/report_manager/assets/banner.jpg



CHAPTER

ONE

CLINICAL KNOWLEDGE GRAPH

version: 1.0

A Python project that allows you to analyse proteomics and clinical data, and integrate and mine knowledge from multiple biomedical databases widely used nowadays.

- Documentation: https://CKG.readthedocs.io
- GitHub: https://github.com/MannLabs/CKG
- Free and open source software: MIT license
- Reference: https://www.biorxiv.org/content/10.1101/2020.05.09.084897v1
- Graph Database dump file and additional relationships: https://data.mendeley.com/datasets/mrcf7f4tc2/1

1.1 Abstract

The promise of precision medicine is to deliver personalized treatment based on the unique physiology of each patient. This concept was fueled by the genomic revolution, but it is now evident that integrating other types of omics data, like proteomics, into the clinical decision-making process will be essential to accomplish precision medicine goals. However, quantity and diversity of biomedical data, and the spread of clinically relevant knowledge across myriad biomedical databases and publications makes this exceptionally difficult. To address this, we developed the Clinical Knowledge Graph (CKG), an open source platform currently comprised of more than 16 million nodes and 220 million relationships to represent relevant experimental data, public databases and the literature. The CKG also incorporates the latest statistical and machine learning algorithms, drastically accelerating analysis and interpretation of typical proteomics workflows. We use several biomarker studies to illustrate how the CKG may support, enrich and accelerate clinical decision-making.

1.2 Cloning and installing

Installation requires >= 80 GB of disk space. See details here.

The setting up of the CKG includes several steps and might take a few hours (if you are building the database from scratch). However, we have prepared documentation and manuals that will guide through every step. To get a copy of the GitHub repository on your local machine, please open a terminal windown and run:

```
$ git clone https://github.com/MannLabs/CKG.git
```

This will create a new folder named "CKG" on your current location. To access the documentation, use the ReadThe-Docs link above, or open the html version stored in the *CKG* folder *CKG/docs/build/html/index.html*. After this, follow the instructions in "First Steps" and "Getting Started".

> **Warning:** If git is not installed in your machine, please follow this tutorial to install it.

1.3 Features

- Cross-platform: Mac, and Linux are officially supported. Instructions for Windows exist.
- Docker container runs all neccessary steps to setup the CKG.

1.4 Disclaimer

This resource is intended for research purposes and must not substitute a doctor's medical judgement or healthcare professional advice.

1.5 Important Note

The databases provided within the Clinical Knowledge Graph (CKG) have their own licenses and the use of CKG still requires compliance with these data use restrictions. Please, visit the data sources directly for more information:

Source type	Source	URL	Reference
Database	UniProt	https://www.uniprot.org/	https://www.ncbi
Database	TISSUES	https://tissues.jensenlab.org/	https://www.ncbi
Database	STRING	https://string-db.org/	https://www.ncbi
Database	STITCH	http://stitch.embl.de/	https://www.ncbi
Database	SMPDB	https://smpdb.ca/	https://www.ncbi
Database	SIGNOR	https://signor.uniroma2.it/	https://www.ncbi
Database	SIDER	http://sideeffects.embl.de/	https://www.ncbi
Database	RefSeq	https://www.ncbi.nlm.nih.gov/refseq/	https://www.ncbi
Database	Reactome	https://reactome.org/	https://www.ncbi
Database	PhosphoSitePlus	https://www.phosphosite.org/	https://www.ncbi
Database	Pfam	https://pfam.xfam.org/	https://www.ncbi
Database	OncoKB	https://www.oncokb.org/	https://www.ncbi
Database	MutationDs	https://www.ebi.ac.uk/intact/resources/datasets#mutationDs	https://www.ncbi
Database	Intact	https://www.ebi.ac.uk/intact/	https://www.ncbi
Database	HPA	https://www.proteinatlas.org/	https://www.ncbi
Database	HMDB	https://hmdb.ca/	https://www.ncbi
Database	HGNC	https://www.genenames.org/	https://www.ncbi
Database	GwasCatalog	https://www.ebi.ac.uk/gwas/	https://www.ncbi
Database	FooDB	https://foodb.ca/	
Database	DrugBank	https://www.drugbank.ca/	https://www.ncbi
Database	DisGeNET	https://www.disgenet.org/	https://www.ncbi
Database	DISEASES	https://diseases.jensenlab.org/	https://www.ncbi
Database	DGIdb	http://www.dgidb.org/	https://www.ncbi
Database	CORUM	https://mips.helmholtz-muenchen.de/corum/	https://www.ncbi
Database	Cancer Genome Interpreter	https://www.cancergenomeinterpreter.org/	https://www.ncbi
Ontology	Disease Ontology	https://disease-ontology.org/	https://www.ncbi
Ontology	Brenda Tissue Ontology	https://www.brenda-enzymes.org/ontology.php?ontology_id=3	https://www.ncbi
Ontology	Experimental Factor Ontology	https://www.ebi.ac.uk/efo/	https://www.ncbi
Ontology	Gene Ontology	http://geneontology.org/	https://www.ncbi
Ontology	Human Phenotype Ontology	https://hpo.jax.org/	https://www.ncbi

Table 1 – continued from previous page

Ontology	SNOMED-CT	http://www.snomed.org/	https://www.ncbi
Ontology	Protein Modification Ontology	https://www.ebi.ac.uk/ols/ontologies/mod	https://www.ncbi
Ontology	Molecular Interactions Ontology	https://www.ebi.ac.uk/ols/ontologies/mi	https://www.ncbi
Ontology	Mass Spectrometry Ontology	https://www.ebi.ac.uk/ols/ontologies/ms	https://www.ncbi
Ontology	Units Ontology	https://bioportal.bioontology.org/ontologies/UO	https://www.ncbi

CHAPTER

TWO

REQUIREMENTS

System Requirements

2.1 System Requirements

The Clinical Knowledge Graph was conceived as a multi-user platform and therefore requires installation in a server-like setup and data systems administration knowledge. However, individual users can have local instances of the CKG, making sure data, software and hardware requirements are fulfilled.

2.1.1 Data

Licensed databases used by the CKG package require login and authentication in order to download their data. This is the case of **SNOMED-CT**, **DrugBank** and **PhosphoSitePlus**. Make sure you sign up to these three databases well in advance as the licensing process can take several days to conclude.

To sign up go to PSP Sign up, DrugBank Sign up and SNOMED-CT Sign up, and follow the instructions.

Once you have been given authorization to access the data, please download the files as follows:

- PhosphoSitePlus: *Acetylation_site_dataset.gz*, *Disease-associated_sites.gz*, *Kinase_Substrate_Dataset.gz*, *Methylation_site_dataset.gz*, *O-GalNAc_site_dataset.gz*, *O-GlcNAc_site_dataset.gz*, *Phosphorylation_site_dataset.gz*, *Regulatory_sites.gz*, *Sumoylation_site_dataset.gz* and *Ubiquitination_site_dataset.gz*.
- DrugBank: *All drugs* (under *COMPLETE DATABASE*) and *DrugBank Vocabulary* (under *OPEN DATA*).
- SNOMED-CT: *Download RF2 Files Now!*.

These files will be later used in Build Neo4j graph database.

2.1.2 Software

- Python 3.7.9
- Redis server
- Neo4j Desktop
- Neo4j database == 4.2.3

2.1.3 Hardware

- Memory: 16Gb
- Disk space: >= 200Gb
- Stable internet connection

Note: When building the Docker image further disk space is needed. We recomend allocating 300Gb, although the final CKG image will be around 150Gb.

CHAPTER

THREE

INSTALLATION

Follow this guide to get the Clinical Knowledge Graph installed. There are two options:

1) Install each of the requirements
2) Build the Docker image and run everything (Neo4j, CKG app and Jupyter Hub) into one single container *(Recommended)*

3.1 Installation

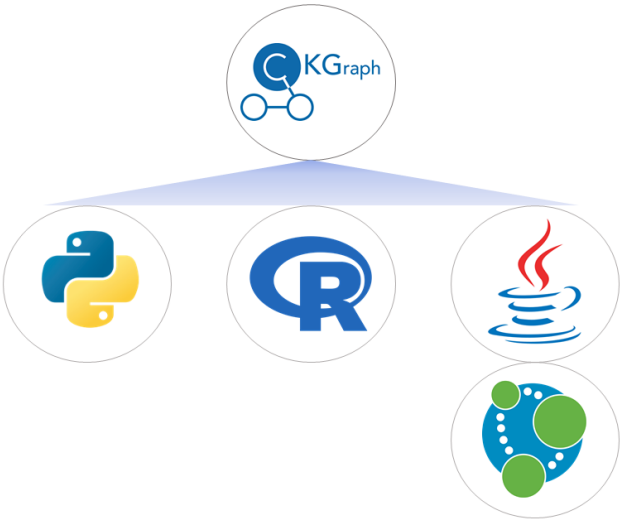
The installation of CKG can be done in two different ways:

1. *Installation of Requirements* –> This installation is recommended especially for admin users that want to customize and configure CKG app, Neo4j database, etc.
2. *Docker Image* –> This installation is the easiest to install and can be used to quickly set up a server version of CKG and its components (Neo4j, JupyterHub). Admin users can still customize these services by modifying how the container is built.

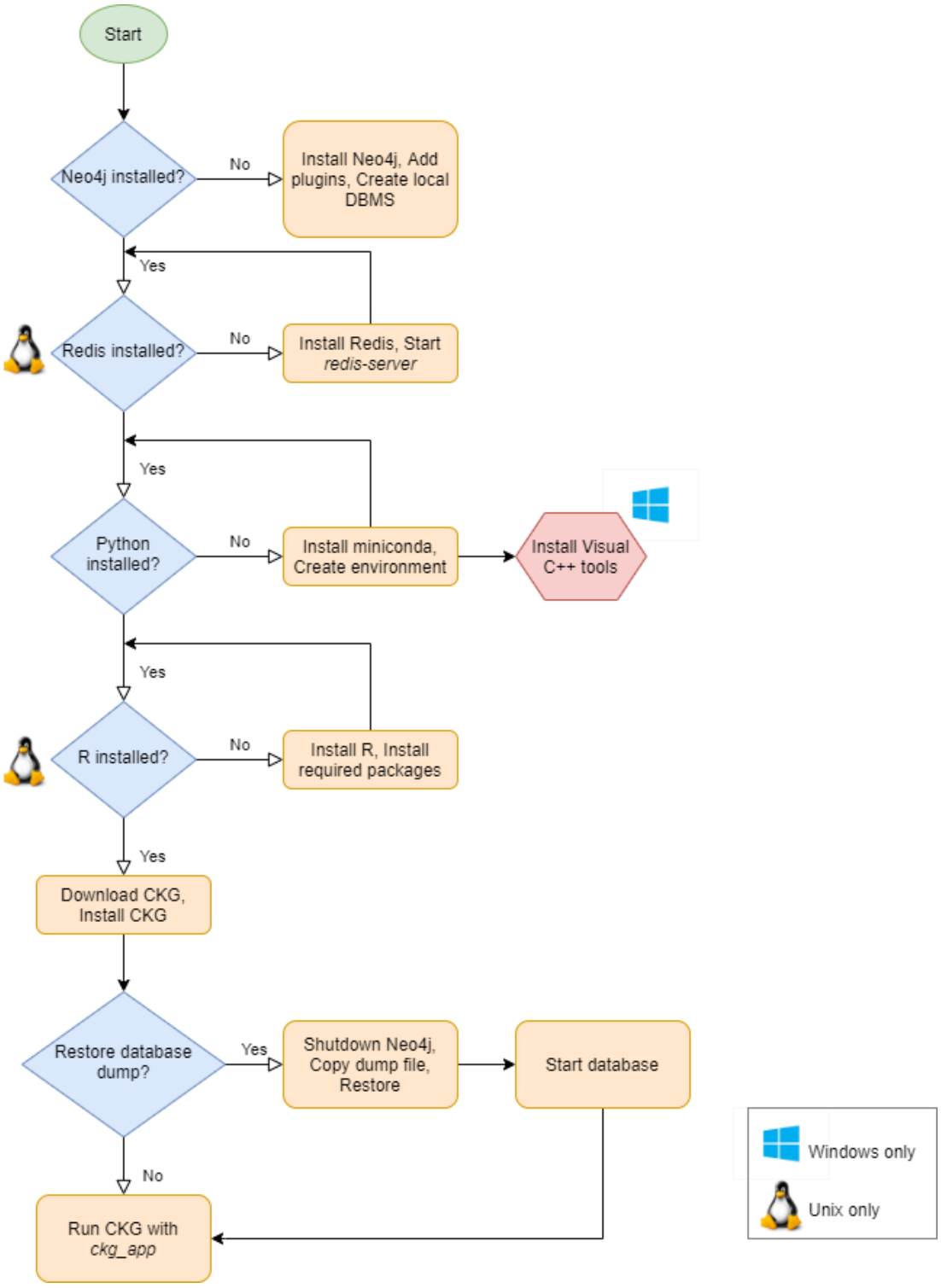
3.1.1 Installation of Requirements

This installation is a two step process:

1. **Installation of Neo4j**: The Neo4j graph database needs to be installed (https://neo4.com) and following the instructions here: *Installing Neo4j*
2. **Installation of CKG's python library**: installation of python and CKG's modules:
    - Python
    - *Installing CKG python library*
    - Optional:
        - R (**Only Unix OS** *for now*)
        - Redis (**Only Unix OS**)



You can follow this flowchart to make sure you have everything you need:



The installation instructions are optimised for operating systems MacOS and Linux. The installation on Windows systems is slightly different, please go to Windows installation.

Requirements

Python

To facillitate the installation of Python, we recommend to use the Miniconda installer:

1. Go to https://docs.conda.io/en/latest/miniconda.html and download the latest installer for your Operating System.
2. Install python following the instructions in the installation wizard
3. Open a terminal window to create a pyhton environment (https://docs.conda.io/projects/conda/en/4.6.1/user-guide/concepts.html#conda-environments)
4. Run `conda create -n ckgenv python= 3.7.9` (ckgenv is the name os the environment)
5. **Activate** the environment by running in the terminal window: `conda activate ckgenv`

Note: To deactivate the environment run: `conda deactivate`

Warning: Always activate `ckgenv environment` when starting CKG app or accessing Jupyter notebooks.

R

Some of the analysis in CKG use R libraries (i.e SAMR, WGCNA) so they require having R installed. This installation is optional and for now compatible only for Unix Operating Systems but we are working on making them available also for Windows. Hence, the installation of R is only required when installing CKG in a Unix OS.

Make sure you have installed **R version >= 3.6.1**:

```
$ R --version
```

And that R is installed in `/usr/local/bin/R`:

```
$ which R
```

To install the necessary R packages, simply initiate R (terminal or shell) and run:

```
install.packages('BiocManager')
BiocManager::install()
BiocManager::install(c('AnnotationDbi', 'GO.db', 'preprocessCore', 'impute'))
install.packages(c('devtools', 'tidyverse', 'flashClust', 'WGCNA', 'samr'),
        dependencies=TRUE, repos='http://cran.rstudio.com/')
install.packages('IRkernel')
```

Note: If you need to install R, follow these tutorial.

Warning: In Mac OS, make sure you have **XQuartz** installed, as well as **Xcode**. For more information on how to install R on OS X, you can follow this link.

Now that you are all set, you can move on and start with Neo4j.

Redis (Only Unix OS)

This installation is only necessary for Unix Operating Systems (i.e MacOS, Linux) since Windows 10 already comes with Redis installed.

edis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. CKG uses redis-server in combination with [Celery queues](https://docs.celeryproject.org/en/stable/getting-started/introduction.html) to run asynchronous tasks such as project creation or project report generation.

For more details on how to install Redis you can follow the instructions [here](https://redis.io/topics/quickstart).

The installation steps are (check Ubuntu installation below):

1) Download Redis

```
$ wget http://download.redis.io/redis-stable.tar.gz
```

2) Untar the downloaded file

```
$ tar xvzf redis-stable.tar.gz
```

3) Install redis using *make*

```
$ cd redis-stable
$ make
```

When running CKG app, you will need to start first the Redis server with:

```
$ redis-server
```

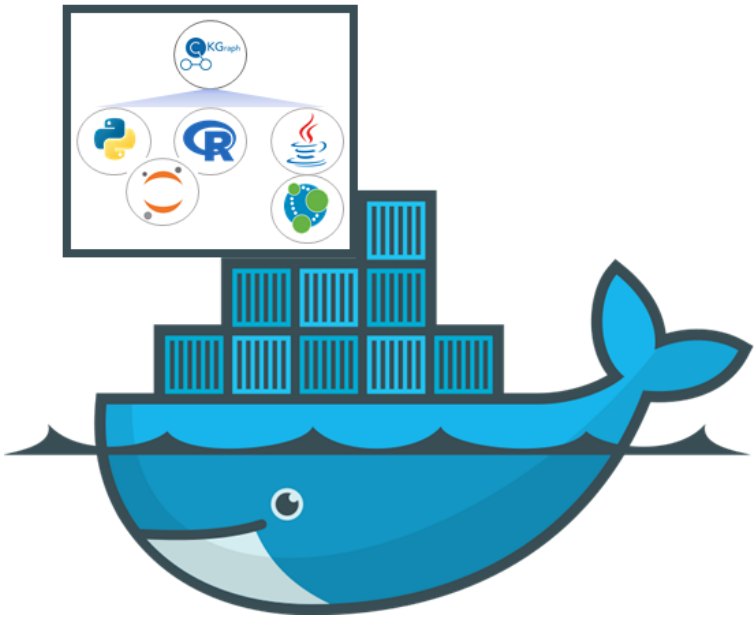
Warning: In Ubuntu, the installation of Redis can be done just with:

$ apt-get install redis-server

3.1.2 Docker Image

To avoid possible compatibility issues caused by different Operating systems, python and R versions, we recommend to follow the instructions to build and use the *CKG Docker Container* instead (https://www.docker.com/). By building the container and running it, you will get:

- Neo4j
- CKG
- JupyterHub



3.2 CKG Docker Container

In this section we describe how to set up the Clinical Knowledge Graph from a Docker container.

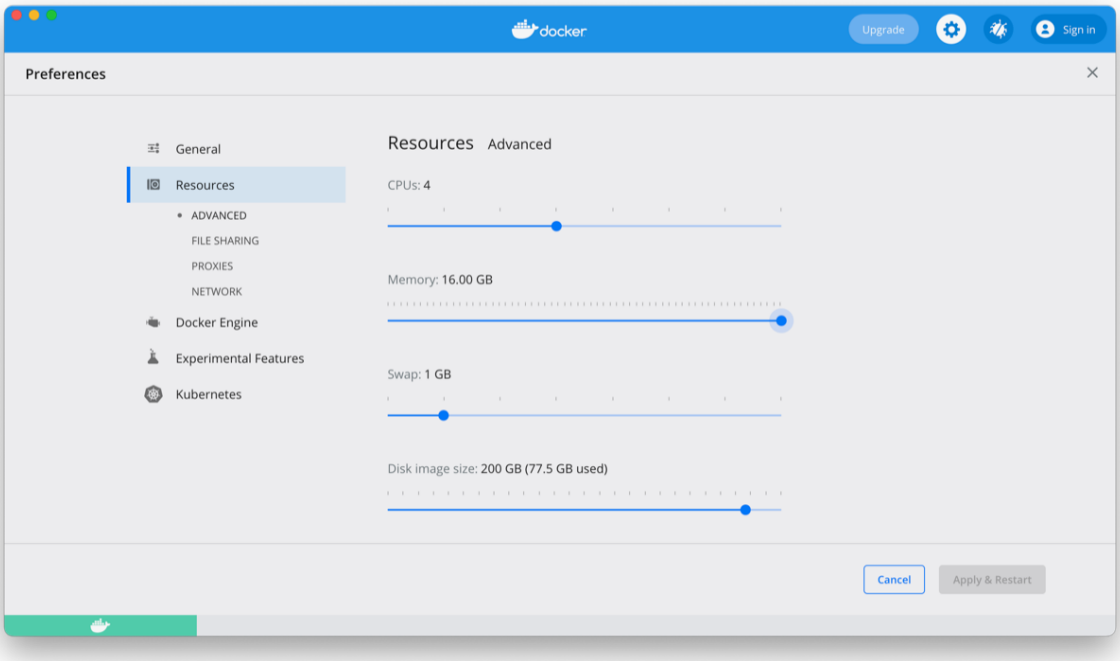
What is Docker? Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production (*source: https://docs.docker.com/get-started/overview/ *).

This container will install all the requirements needed, download source databases and build the CKG graph database, and open 5 ports through which to interact with the CKG.

Warning: This requires Docker to be previously installed (https://docs.docker.com/engine/install/). Notice that the installation in Unix systems is a bit different than in Windows or MacOS, check the instructions for your specific distribution (https://docs.docker.com/engine/install/#server).

Warning: Building the container requires **~200Gb of disk space**. The image takes ~150Gb but during the building process you will need around 200Gb of disk space available (loading the database dump takes quite a bit of disk temporarily).

Warning: In MacOS, you will need to allocate more resources to Docker to adapt to CKG's requirements. You can do this in Docker desktop -> Preferences. We recommend (at least) the following configuration:

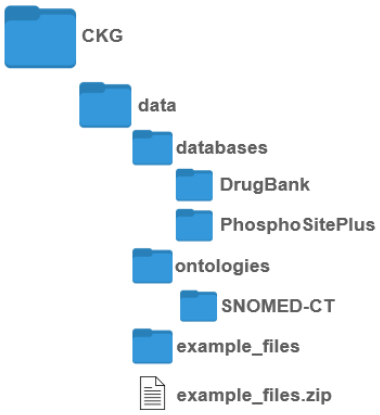


To run the Docker, simply:

1. Clone CKG's repository: https://github.com/MannLabs/CKG
2. Get licensed databases: *Licensed Databases*
3. Copy the licensed databases in the right directories:

- CKG/data/databases/{DrugBank|PhosphoSitePlus}
- CKG/data/ontologies/SNOMED-CT

The data folder will look like this afterwards:



4. Build thecontainer:

```
$ cd CKG/
$ docker build -t docker-ckg:latest .
```

5. Run docker:

```
$ docker run -d --name ckgapp -p 7474:7474 -p 7687:7687 -p 8090:8090 -p 8050:8050␣
↪docker-ckg:latest
```

Once the container is running, you can open an interactive bash session using:

```
$ docker exec -it ckgapp bash
```

You will have access to the logs within the container at:

- **Neo4j**: `/var/log/neo4j`
- **CKG**: `/CKG/log`
- **uwsgi**: `/var/log/uwsgi`

Once the docker is running:

1. Access JupyterHub: http://localhost:8090/:

- user:ckguser
- password:ckguser

2. Access Neo4j browser (connection may take several minutes): http://localhost:7474/

   Login using:

   - user: neo4j
   - password: NeO4J

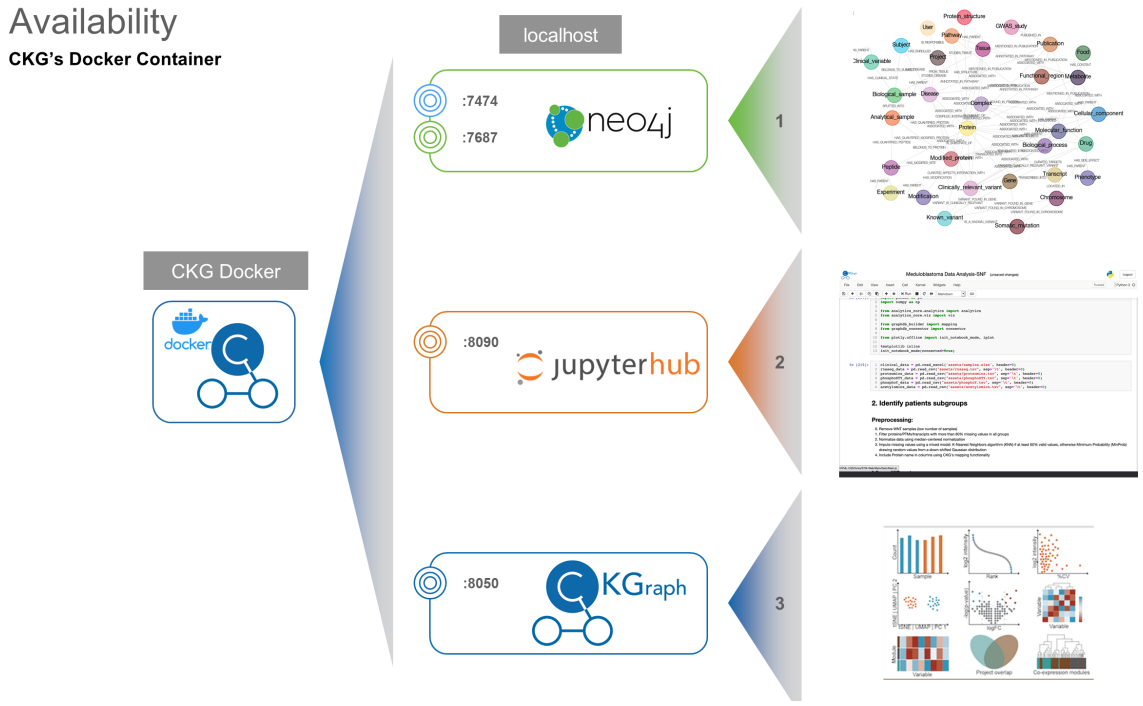
When the database is running:

1. In your web browser access CKG app: http://localhost:8050/

   Login using the test user:

   - user: test_user
   - password: test_user

2. In the Home page navigate to the Admin page
3. Run Minimal update (*Minimal update*) (these can take a while but will run in the background. Follow progress in the docker dashboard logs) and create a user
4. Explore options in CKG



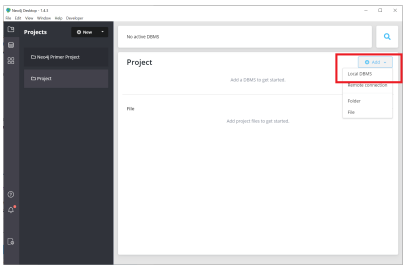
Note: Remember that with Docker Desktop (https://www.docker.com/products/docker-desktop), you can check the logs of the running image.

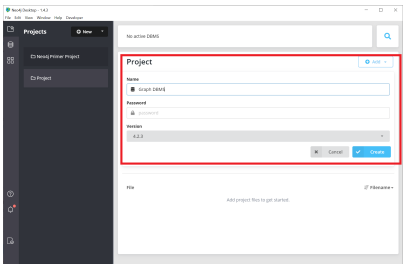
3.3 Installing Neo4j

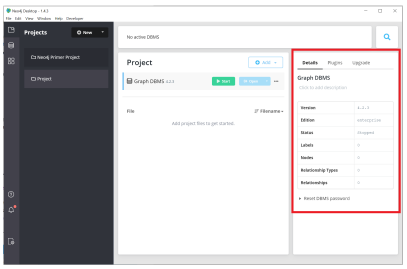
Getting started with Neo4j is easy, just install Neo4j desktop and you are ready.

3.3.1 Neo4j Desktop (Windows/MacOS)

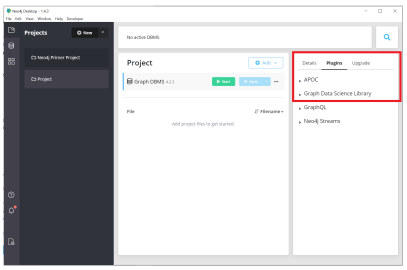
You can download Neo4j Desktop from the Neo4j download page. Neo4j Desktop is bundled with a Java so you won't need to install Java separately. If you decide to install Neo4j directly, please check the requirements here. The Community Edition of the software is free but a sign up is required. Once the file has downloaded, you can install Neo4j by following the instructions automatically opened in the browser.

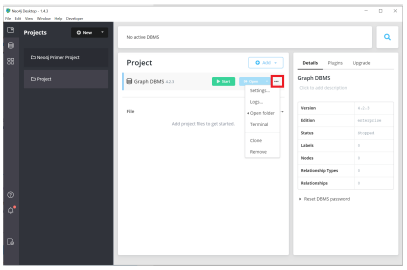


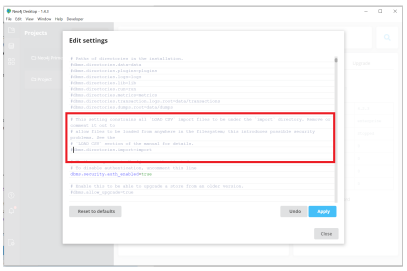




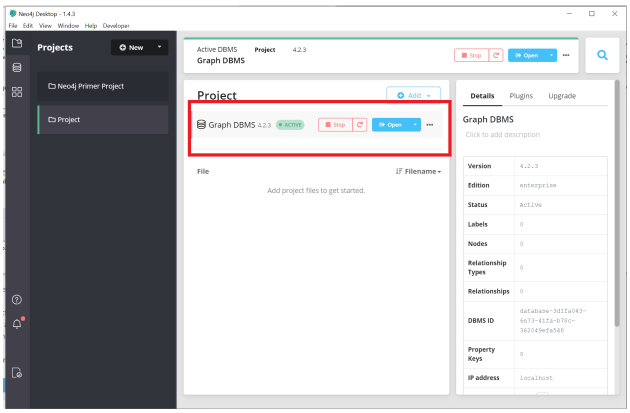
Open the Neo4j Desktop App and create a database by clicking *Add*, followed by *Local DBMS*, **choose database version 4.2.3** using the password "NeO4J". Now that your database is created:







1. Click *Manage* and then *Plugins*. Install "**APOC**" and "**Graph Data Science Library**".
2. Click the tab *Settings*, and comment the option `dbms.directories.import=import` by adding `#` at the beginning of the line and set the name of the database to `graph.db` by editing the line `dbms.default_database` to `dbms.default_database=graph.db`.
3. Click *Apply* at the bottom of the window.
4. Start the Graph by clicking the play sign, at the top of the window.



If the database starts and no errors are reported in the tab *Logs*, you are ready go to!

3.3.2 Neo4j Desktop (Unix)

The installation in Unix systems is a little bit different. Follow the specific instructions for your distribution here: https://neo4j.com/docs/operations-manual/current/installation/linux/

To get Neo4j 4.2.3 and the Graph Data Science and APOC plugins in the Docker container, we use the following instructions:

```
# Installation openJDK 11
add-apt-repository ppa:openjdk-r/ppa
apt-get update
apt-get install -yq openjdk-11-jdk

# NEO4J 4.2.3
wget -O - https://debian.neo4j.com/neotechnology.gpg.key | apt-key add - && \
echo "deb [trusted=yes] https://debian.neo4j.com stable 4.2" > /etc/apt/sources.list.
↪d/neo4j.list && \
apt-get update && \
apt-get install -yq neo4j=1:4.2.3

## Setup initial user Neo4j
rm -f /var/lib/neo4j/data/dbms/auth && \
   neo4j-admin set-initial-password "NeO4J"

## Install graph data science library and APOC
RUN wget -P /var/lib/neo4j/plugins https://github.com/neo4j/graph-data-science/
↪releases/download/1.5.1/neo4j-graph-data-science-1.5.1.jar
RUN wget -P /var/lib/neo4j/plugins https://github.com/neo4j-contrib/neo4j-apoc-
↪procedures/releases/download/4.2.0.4/apoc-4.2.0.4-all.jar
```

3.4 Installing CKG python library

Setting up the Clinical Knowledge Graph is straightforward. Assuming you have **Python 3.7.9** already installed and a virtual environment created following instructions here: *Installation*.

3.4.1 Setting up the Clinical Knowledge Graph

The first step in setting up the CKG, is to obtain the complete code by cloning the GitHub repository:

```
$ git clone https://github.com/MannLabs/CKG.git
```

Another option is to download it from the github page directly:

1. Go to https://github.com/MannLabs/CKG
2. In *Code* select **Download ZIP**
3. Unzip the file

Once this the cloning is finished or the file is unzipped, you can install CKG by running:

```
$ cd CKG/
$ conda activate ckgenv
$ python setup.py install
```

This will automatically create the `data` folder and all subfolders, as well as setup the configuration for the log files where all errors and warnings related to the code will be written to. Further, it will create an executable file with CKG's app. To start the app, simpy run:

```
$ ckg_app
```

Warning: If you are using a Unix Operating System (i.e MacOS or Linux), you will need to start Redis server by running: $ redis-server

3.5 Create a new user in the graph database

Warning: CKG has implemented a simple authentication and user management method, but industry-accepted platforms, vendors, or libraries should be used in a production setup.

The creation of a new user includes two steps:

1. The user who is currently logged-in in the database and invoking the commands, has to add the new user to the system and attribute it a role, by default `reader`.
2. Each user is added in the graph database as a new `User` node, with attributes: id, username, name, acronym, email, secondary email, phone number, affiliation, rolename and expiration date.

There are multiple ways to create a new user:

From the command line: *(one user at a time)*

```
$ cd ckg/graphdb_builder/builder
$ python create_user.py -u username -d password -n name -e email -s second_email -p␣
↪phone_number -a affiliation
```

From an excel file: *(multiple users)*

```
$ cd ckg/graphdb_builder/builder
$ python create_user.py -f path/to/excel/file
```

For help on how to use `create_user.py`, run:

```
$ python create_user.py -h
```

Warning: If you want to have spaces (" ") in any of the arguments (e.g. -n name), you need to have the argument value within quotes "" (e.g. -n "John Smith"). The same applies to other arguments like affiliation.

From an CKG's app:

In CKG's app there is an Admin section that provides the option to create users in the database.

Note: When creating a new user through the app, the initial password is the samne as the username.

CHAPTER

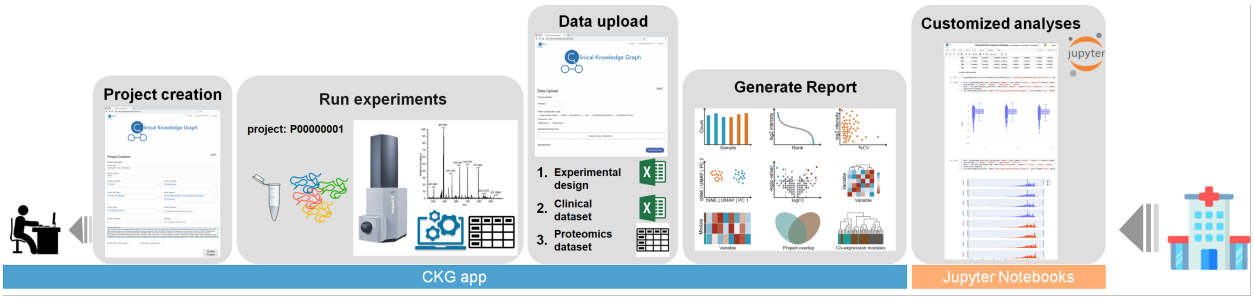
FOUR

GETTING STARTED

Are you new to the Clinical Knowledge Graph? Learn about how to use it and all the possibilities.

- **Creating a project and getting a report out (default configuration)** *Getting Started with CKG*
- **Connecting to CKG Graph Database**: *Connect to DB*
- **Define a different Analysis Pipeline**: *Configuration*
- **Report Notification**: *Notifications*

4.1 Getting Started with CKG



4.1.1 1. Project creation

4.1.2 2. Run experiments

4.1.3 3. Data upload

4.1.4 4. Generate Report

4.1.5 5. Customize analyses

4.2 Connecting to the Clinical Knowledge Graph database

In order to make use of the CKG database you just built, we need to connect to it and be able to query for data. This connection is established via Neo4j's Python driver `neo4j <https://neo4j.com/docs/api/python-driver/current/api.html/>`, a library and comprehensive toolkit developed to enable working with Neo4j from within Python applications, and should already be installed in your virtual environment.

Another essential tool when working with Neo4j databases, is the Cypher query language. We recommend becoming familiar with it, to understand the queries used in the different analyses.

4.2.1 Neo4j connector

Note that this section is for illustration purposes only.

Within the CKG package, the `graph_connector` module was created to connect the different parts of the Python code, to the Neo4j database and allow their interaction.

In this module, the *Graph* class from `Neo4j` is used to represent the graph data storage space within the Neo4j database, and a YAML configuration file is parsed to retrieve the connection details. The configuration file `connector_config.yml` contains the database server host name, server port, user to authenticate as, and password to use for authentication.

```
from ckg.graphdb_connector import connector
driver = connector.getGraphDatabaseConnectionConfiguration()
```

Once the connection is established, the database can be queried. For example:

```
example_query = 'MATCH (p:Project)-[:HAS_ENROLLED]-(s:Subject) RETURN p.id as project_
↪id, COUNT(s) as n_subjects'
results = connector.getCursorData(driver=driver, query=example_query, parameters={})
```

This query searches the database for all the available projects and counts how many subjects have been enrolled in each one, returning a pandas DataFrame with "project_id" and "n_subjects" as columns.

4.2.2 Changing/Updating database connection

The connection to the graph database requires credentials, which are stored in `graphdb_connector/connector_config.yml`. This files includes the following lines:

```
db_url: "0.0.0.0"
#dbPort = 7688 #Production environment
db_port: 7687 #Test environment
db_user: "neo4j"
db_password: "NeO4J"
```

The initial password to create a new Neo4j database is set to **NeO4J**. If you would like to use another password when creating the database, you can edit the mentioned file and replace **NeO4J** with any other password of your choosing. Another option is to change the password directly in the database by accessing *Manage* in the Neo4j desktop window, select the tab *Administration* and then set the new password. Ultimately, make sure that the password in `graphdb_connector/connector_config.yml` and in the Neo4j database are the same.

4.3 Define data analysis parameters

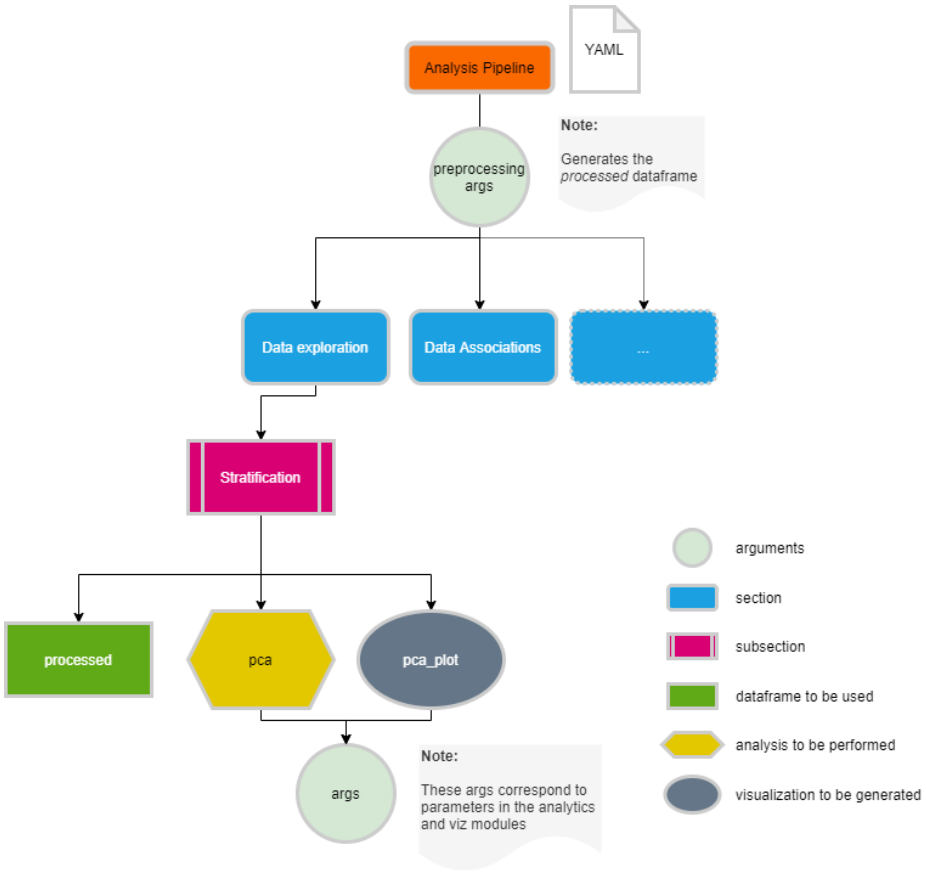
A multitude of different analysis methods and visualisation plots have been implemented within the `analytics_core` of the Clinical Knowledge Graph. The table below contains the current list of methods and visualizations available:

Table CKG Analytics Core

	Step	Method	Description
Data preparation	Power analysis	anova power analysis	Determine the sample size required to det
	Filtering	percentage	Filtering based on maximum percentage c
		at_least_x	Filtering based on minimum number of pr
	Imputation	K-nearest Neighbors	Imputation based on the algorithm Neares
		Probabilistic Minimum Imputation approach	Imputation method replacing missing valu
		Mixed model	A combination of KNN and Probabilistic
	Normalization	Median	Normalize samples using the median
		Median polish	Normalization based on the medians obta
		Quantile	Adjustment method that forces the observ
		Linear	Apply l1 or l2 normalization
	Batch effect correction	COMBAT	Adjust for batch effects in datasets where
Data exploration	Ranking	Ranking	Ranking of proteins based on intensity
	Coefficient of Variation	Coefficient of Variation	Coefficient of variation per group as a qua
	QC markers	QC Makers	If there are quality control markers associ
	Summary statistics	Summary statistics	Statistics for rows and columns in the data
Data analysis	Dimensionality reduction	PCA	Principal Component Analysis (2D, 3D)
		tSNE	t-distributed Stochastic Neighbor Embedd
		UMAP	Uniform Manifold Approximation and Pr
	Hypothesis testing	SAMR	Significance analysis of microarrays appli
		ANOVA	Analysis of Variance
		ANOVA-rm	Analysis of Variance for repeated measure
		t-test	t-test mean difference
	Multiple-test correction	Bonferroni	Bonferroni p-value correction
		Benjamini-Hochberg	Benjamini-Hochberg FDR correction
		Permutation-FDR	Permutation FDR
		...	...
	Correlation	Correlation	Pearson or Spearman correlation
		Correlation-rm	Pearson or Spearman correlation for repea
	Enrichment	Single Sample Gene Set Enrichment Analysis	Single-sample GSEA (ssGSEA), an exten
		Fisher Exact Test	Significant test for contingency tables
	Network analysis	Louvain partition	Best partition algorithm
		Greedy modularity	A hierarchical agglomeration algorithm fo
		Asynchronous label propagation algorithm	The algorithm initializes each node with a
		Girvan-Newman algorithm	Hierachical algorithm that removes edges
		Affinity propagation	A centroid-based clustering algorithm tha
		Similarity Network Fusion	SNF computes and fuses patient similarity
	Multiomics	WGCNA	Weighted gene co-expression network ana
Visualization	Viz	Pie chart	Circular statistical chart, which is divided
		Distribution plot	Representations of statistical distributions
		Bar chart	
		Scatter plot matrix	
		Ranking plot	
		Scatter plot	
		Volcano plot	
		Heatmap plot	
		Heatmap plot with annotation and clustering	
		Network	Generates a Cytoscape network (Plot.ly),
		PCA plot	PCA plot with loadings (2D and 3D)
		Sankey diagram	Visualize the contributions to a flow
		Table	
		Violin plot	

		Parallel coordinates plot	
		WGCNA plots	Generates all the plots for the WGCNA a
		2-way Venn diagram	
		Word cloud	Represents the frequency of words in a te
		Save Dash plot	Save a Dash figure object to svg format
		Kaplan-meier plot	Kaplan-meier survival plot with significa
		Polar chart	Represents data along radial and angular

The default workflow makes use of the functions defined in this module and runs, for each data type, the analysis pipeline defined in a configuration file. These configuration files are defined in YAML format (https://yaml.org/spec/1.2/spec.html), which can be easily read in Python into a dictionary structure with sections and analyses. For each analysis we need to define the data that will be used (i.e original data), how the results will be visualized (i.e pca_plot) and what parameters need to be used (i.e components: 2).



In the CKG, we have default analyses defined for clinical, proteomics, phosphoproteomics and interactomics datasets. All the analysis configuration files can be modified to fit your project or data. To check how each configuration files look like and how to modify them, please follow the links below and also review the specific functions in CKG's *API Reference* to define the args to use.

4.3.1 Clinical data analysis parameters

Currently, the clinical default analysis pipeline can be accessed here.

The Clinical data configuration file contains two sections: `args` and `overview`. The first section contains the parameters used for the processing of the raw clinical data. To obtain the raw clinical data, we query the CKG database for all the clinical variables connected to biological samples in a specific project. This results in a Pandas dataframe with all the relevant information. To process the raw data, a number of parameters are defined in the `args` section of the configuration file:

Fig. 1: Clinical Data configuration file

- **subject_id**: column label containing subject identifiers.
- **sample_id**: column label containing biological sample identifiers.
- **group_id**: column label containing group identifiers.
- **imputation_method**: method for missing values imputation ("KNN", "distribuition", or "mixed").
- **columns**: list of column names whose unique values will become the new column names
- **values**: column label containing clinical variable values.
- **extra**: additional column labels to be kept as columns

The result is another Pandas dataframe, stored as "processed", where columns are the clinical variables and biological samples are rows, group and subject identifier are kept as columns as well.

Note: We advise to change only **imputation_method**, if needed.

The second section (`overview`) depicts the analysis performed for the clinical data, and the parameters used to do it. Among the analysis is:

- Summary table (**clinical variables**)
- Stratification plot (**stratification**)
- Clinical variables per group (**measurement matrix**)
- Hypothesis test (**regulation**)
- Correlation network (**correlation**)

Within each analysis, specific parameters are defined:

- **description**: Definition of the analysis used.
- **data**: defines on which dataset dataframe the analysis will be ran (e.g. "clinical variables", "original", "processed").
- **analyses**: which statistical analysis to run on the `data`. These functions are called from the mocdule `analytics_factory.py`.
- **plots**: which plot to use to show the results of **analyses**. Functions also called from the mocdule `analytics_factory.py`.
- **store_analysis**: boolean. True if the dataframe resulting from **analyses** is to be stored.
- **args**: all arguments necessary for **analyses** and **plots**.

You can modify the analysis parameters just by changing the respective parameters within the configuration file. Remember to consult the modules `analytics.py` and `viz.py`, to learn more about the arguments of each function.

4.3.2 Proteomics data analysis parameters

Similarly to the *Clinical data analysis parameters*, the Proteomics configuration file is divided into sections: `args`, `overview`, `data exploration`, `data associations` and `enrichment`.

The proteomics default analysis pipeline can be accessed here.

The `args` section contains the parameters used to process the proteomics data extracted from the CKG database, which includes the filtering of proteins according to the determined threshold, as well as the imputation of all missing values. These parameters include:

- **imputation**: boolean. Set to *True* if missing values shall be imputed.
- **imputation_method**: method for missing values imputation ("KNN", "distribuition", or "mixed").
- **missing_method**:
- **missing_per_group**: boolean. If *True*, proteins are filtered based on valid values per group; if *False* filter across all samples.
- **missing_max**: maximum ratio of missing/valid values to be filtered. (e.g. **0.3** filters all proteins with more than 30% missing values).
- **min_valid**: minimum number of required valid values to keep a protein.
- **value_col**: column label containing expression values.
- **index**: column labels to be be kept as index identifiers.

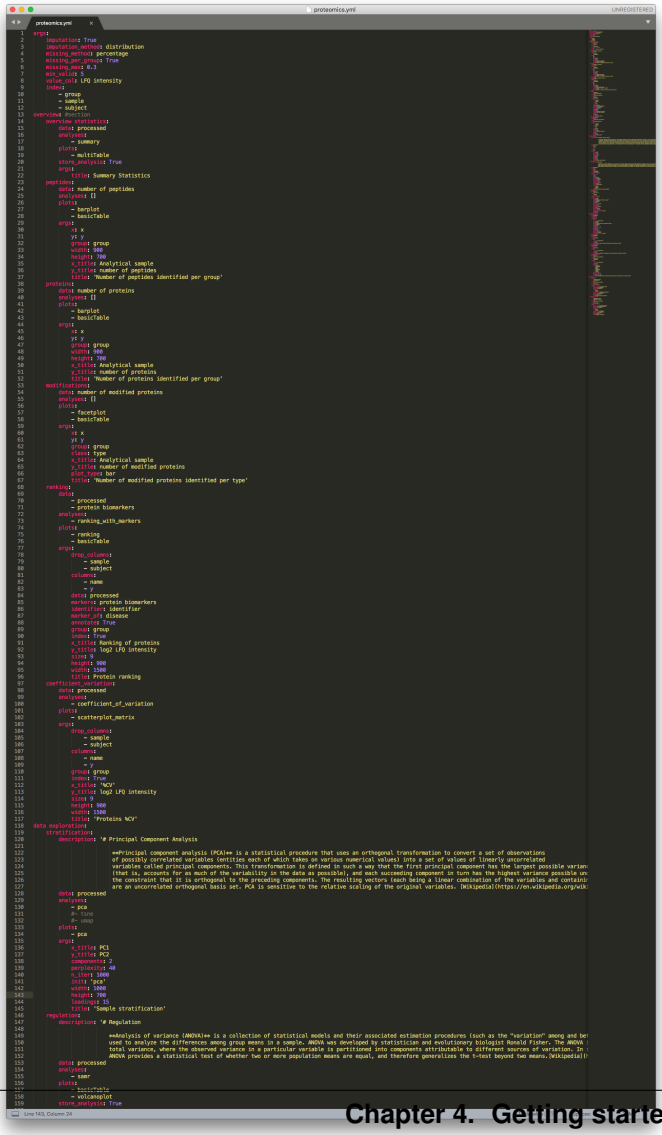


Fig. 2: Proteomics configuration file

- **batch_correction**: boolean. If *True*, CKG will use adjust for batch effect
- **batch_col**: column to be used as batch identifier (*batch* as default)

The result is a Pandas dataframe, stored as "processed", where columns are protein identifiers (UniprotID~GeneName) and analytical samples are rows, group and subject identifier are kept as columns as well.

Note: We advise to change only **imputation**, **imputation_method**, **missing_method**, **missing_per_group**, and **missing_max** or **min_valid**.

The second section (`overview`) corresponds to basic statistics and includes:

- Data summary statistics (**overview statistics**)
- Number of peptides per sample (**peptides**)
- Number of proteins per sample (**proteins**)
- Number of modifications per sample (**modifications**)
- Dynamic range per group (**ranking**)
- Coefficient of variation per group (**coefficient_variation**)

In the `data exploration` section, we look at the sample stratification (**stratification**), differentially expressed proteins (**regulation**), and protein-protein correlation network (**correlation**). In this section, you can choose to modify parameters like `alpha` (FDR value), `s0` (artificial within groups variance) or `fc` (fold-change), to better fit your data and experimental design. Likewise, the correlation network is, by default, set to show correlations above 50% (`cutoff: 0.5`), this too can be changed.

The `data associations` section includes analyses that correlated differentially expressed proteins to proteins (**interaction_network**), drugs (**drug_associations**), diseases (**disease_associations**) and publications (**literature_associations**). All these associations are directly queried from the CKG database.

The last section in the default analysis pipeline corresponds to the `enrichment` analysis, and includes both Gene Ontology (**go_enrichment**) and Pathway (**pathway_enrichment**) enrichment analyses, using Fisher's exact test (`method: fisher`)

Likewise in the *Clinical data analysis parameters*, within each analysis, specific parameters are defined:

- **description**: Definition of the analysis used.
- **data**: defines on which dataset dataframe the analysis will be ran (e.g. "clinical variables", "original", "processed").
- **analyses**: which statistical analysis to run on the `data`. These functions are called from the mocdule `analytics_factory.py`.
- **plots**: which plot to use to show the results of **analyses**. Functions also called from the mocdule `analytics_factory.py`.
- **store_analysis**: boolean. True if the dataframe resulting from **analyses** is to be stored.
- **args**: all arguments necessary for **analyses** and **plots**.

You can modify the analysis parameters just by changing the respective parameters within the configuration file. Remember to consult the modules `analytics.py` and `viz.py`, to learn more about the arguments of each function.

4.3.3 Multiomics data analysis parameters

The Multiomics configuration file allows you to integrate different data types and, as a default we integrate clinical and proteomics data. In this context, the configuration file includes a multi-correlation section where all clinical and proteomics data are correlated and depicted as a network. Another option is to run a Weighted Gene Co-expression Network Analysis (WGCNA), where features (proteins) are clustered in co-expression modules and further correlated to the clinical variables.

The multiomics default analysis pipeline can be accessed here.

4.3.4 Adding New Analyses or visualizations

If you would like to contribute to CKG and add new analysis or visualization functions you can implement them in the *Analytics* or *Viz* modules respectively. To make your new analysis or visualization available as part of the default analytical pipeline, you will need to add a conditional block in the *analytics_factory.py* module. For instance:

```
elif self.
↪analysis_type == "NEW_ANALYSIS":
   arg1 = 'VALUE'
   arg2 = 'VALUE'
   arg3 = 'VALUE'
```

(continues on next page)

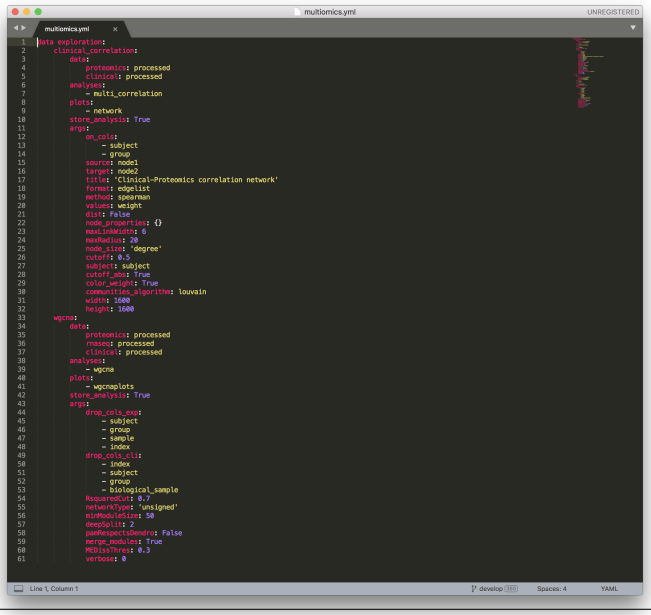


Fig. 4: Multiomics configuration file

(continued from previous page)

```
   if "arg1" in self.args:
      arg1 = self.args["arg1"]
   if "arg2" in self.args:
      arg2 = self.args["arg2"]
   if "arg3" in self.args:
      arg3 = self.args["arg3"]
   self.result[self.
↪analysis_type] = analytics.
↪run_new_function(self.data,
↪ arg1=arg1, arg2=arg2, arg3=arg3)
```

To incorporate this new analysis into the configuration file just add a new section, for example:

```
new analysis section:
   new analysis subsection:
      description: 'This is a new␣
↪function in the analytics core'
      data: processed
      analyses:
         - NEW_ANALYSIS
      plots:
         - scatterplot
      args:
         arg1: 'value1'
         arg2: 'value2'
         arg3: 'value3'
         x_title: 'x axis'
         y_title: 'y axis'
         width: 1000
         height: 700
         title:␣
↪'Scatter plot for my new analysis'
```

Finally, just add the new function to the table here in the docs :)!

4.4 Report notifications

One of the biggest advantages of the standard analysis workflow of the Clinical Knowledge Graph is its speed. However, and depending on your experimental design, number od samples, quantified proteins, etc, this might take several minutes. To make it easier on the user's, and avoid waiting time in front of the screen,

4.4.1 Slack notifications

4.4.2 E-mail notifications

CHAPTER

FIVE

CKG GRAPH DATABASE BUILDER

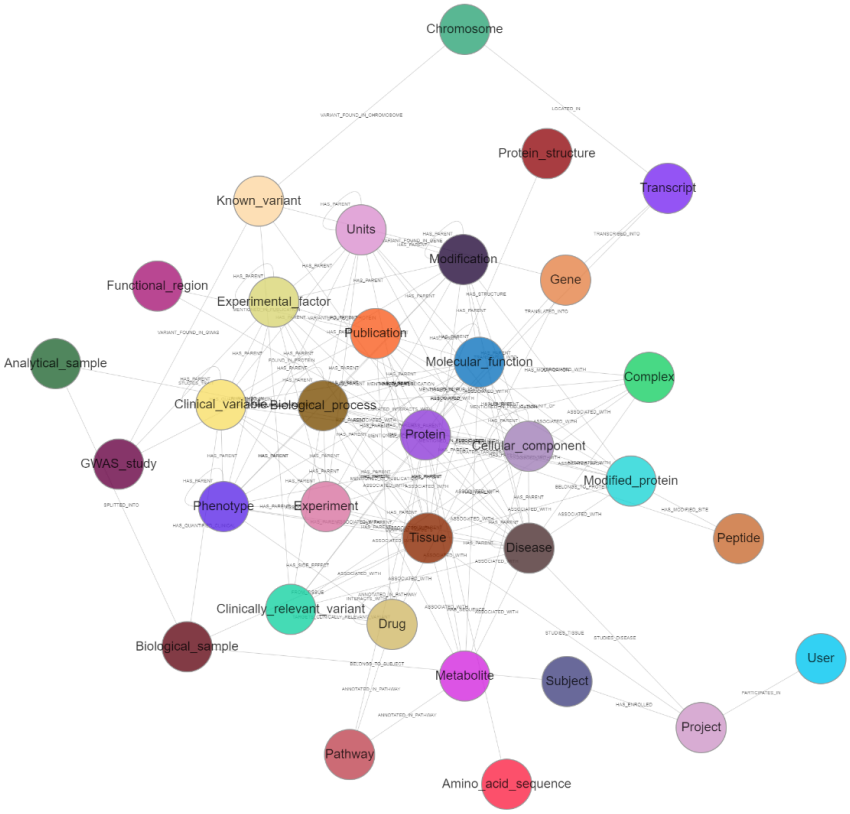
- **Building CKG's Graph Database**: *Builder*
- **Adding New Resources**: *Contributing*

5.1 Building CKG's Graph Database

CKG has a dedicated module (graphdb_builder) that can be used to generate the entire Knowledge Graph (full update) or to update specific ontologies, databases or experiments (partial update). The module works as a 2-step process:

1) Import: uses specific parsers and configuration files to convert ontologies, databases and experimental data (projects) into tab-separated values files with the entities (nodes) and relationships to be imported into the graph database. The generated files are stored in the *data/imports* directory.

2) Loading: When loading data into the graph, we need to specify the type of node (entity) to be loaded. The types of nodes correspond to the defined data model:



The list of entities to be loaded into the graph is defined in the builder configuration file under graph.

CKG has predefined Cypher queries to load the generated tsv files into the graph database. To facilitate the use and understanding of the queries, they are defined in YAML format, which allows to define attributes such as query name, description, nodes and relationships involved.

Warning: Remember that the graph database needs to be running when the database is being built or updated (Loading step).

5.1.1 Licensed Databases

Most of the biomedical databases and ontology files will automatically be downloaded during building of the database. However, the following licensed databases have to be downloaded manually.

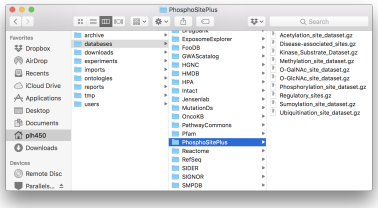


Fig. 1: PhosphoSitePlus database folder.

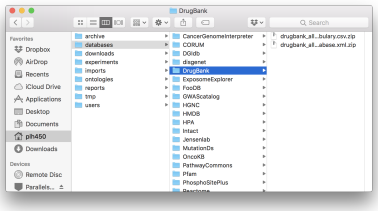


Fig. 2: DrugBank database folder.

´ PhosphoSitePlus: *Acetylation_site_dataset.gz*, *Disease-associated_sites.gz*, *Kinase_Substrate_Dataset.gz*, *Methylation_site_dataset.gz*, *O-GalNAc_site_dataset.gz*, *O-GlcNAc_site_dataset.gz*, *Phosphorylation_site_dataset.gz*, *Regulatory_sites.gz*, *Sumoylation_site_dataset.gz* and *Ubiquitination_site_dataset.gz*. DrugBank: *All drugs* (under *COMPLETE DATABASE*) and *DrugBank Vocabulary* (under *OPEN DATA*). SNOMED-CT: *Download RF2 Files!*.

After download, move the files to their respective folders:

- PhosphoSitePlus: `CKG/data/databases/PhosphoSitePlus`
- DrugBank: `CKG/data/databases/DrugBank`
- SNOMED-CT: `CKG/data/ontologies/SNOMED-CT`

In the case of SNOMED-CT, unzip the downloaded file and copy all the subfolders and files to the `SNOMED-CT` folder.

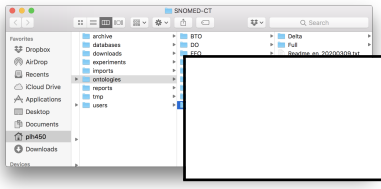
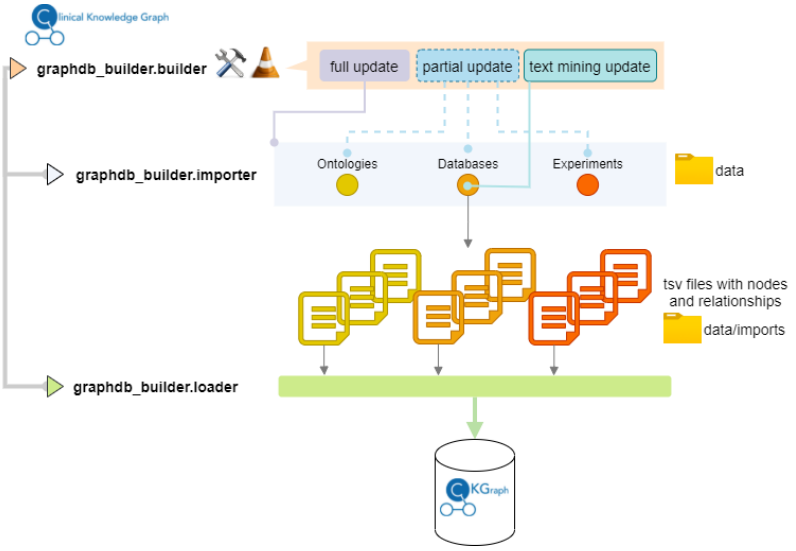


Fig. 3: SNOMED-CT ontology folder.

Warning: These three databases require login and authentication. To si
up go to PSP Sign up, DrugBank Sign up and SNOMED-CT Sign up. In
case of SNOMED-CT, the UMLS license can take several business days.

Note: If the respective database folder is not created, please do it manually.

5.1.2 Graph Database Builder



Full update

The full update goes through both steps (import and loading) and updates all ontologies, databases and the available experiments. There are several options to run the full update:

1) Command-line (executable) *(Recommended)*

```
$ ckg_build
```

This will initiate the full update with default parameters:

- *download=True* – ontologies and databases will be downloaded from their sources
- *n_jobs=3* – 3 processes will be use simultaneously

2) Command-line (programmatically)

The module graphdb_builder/builder/builder.py can be called as a python script in the command-line. Use *-h* to get help on how to use it:

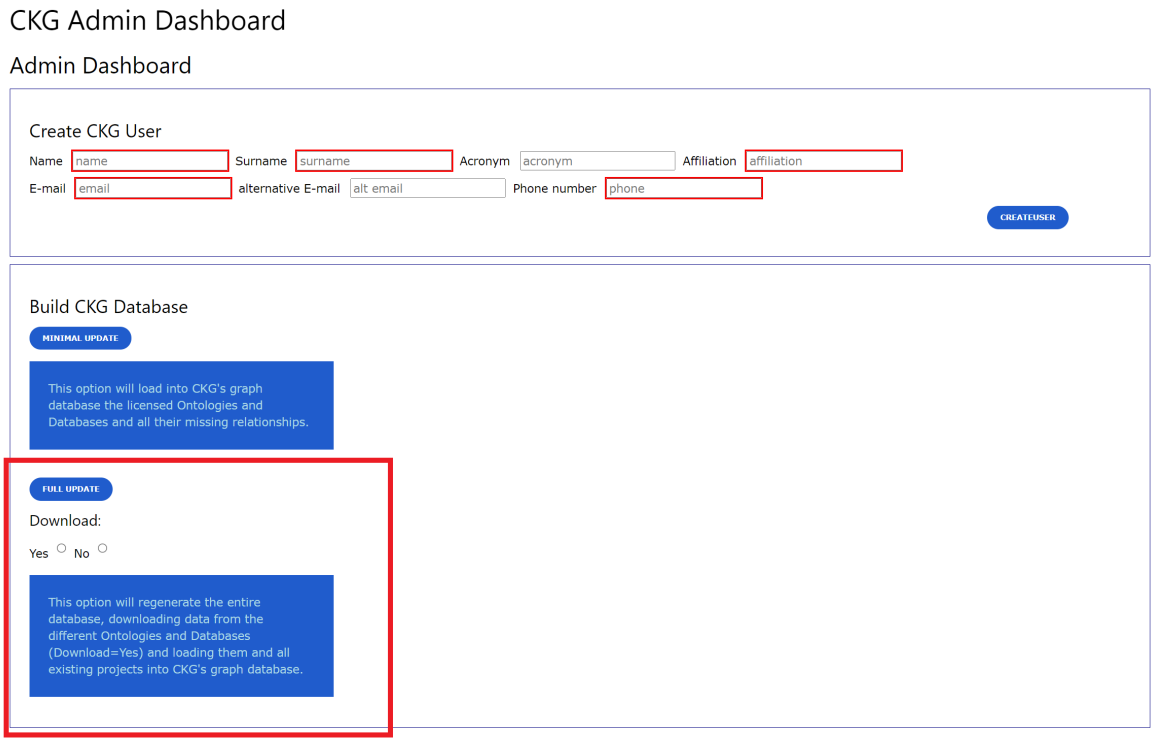
```
$ python builder.py -h

    usage: builder.py [-h] [-b {import,load,full,minimal}]
            [-i {experiments,databases,ontologies,users} [{experiments,databases,
↪ontologies,users} ...]]
            [-l LOAD_ENTITIES [LOAD_ENTITIES ...]] [-d DATA [DATA ...]]
            [-s SPECIFIC [SPECIFIC ...]] [-n N_JOBS] [-w DOWNLOAD] -u
            USER

    optional arguments:
    -h, --help            show this help message and exit
    -b {import,load,full,minimal}, --build_type {import,load,full,minimal}
                            define the type of build you want (import, load, full
                            or minimal (after dump file))
    -i {experiments,databases,ontologies,users} [{experiments,databases,ontologies,
↪users} ...], --import_types {experiments,databases,ontologies,users} [{experiments,
↪databases,ontologies,users} ...]
                            If only import, define which data types (ontologies,
                            experiments, databases, users) you want to import
                            (partial import)
    -l LOAD_ENTITIES [LOAD_ENTITIES ...], --load_entities LOAD_ENTITIES [LOAD_
↪ENTITIES ...]
                            If only load, define which entities you want to load
                            into the database (partial load)
    -d DATA [DATA ...], --data DATA [DATA ...]
                            If only import, define which ontology/ies,
                            experiment/s or database/s you want to import
    -s SPECIFIC [SPECIFIC ...], --specific SPECIFIC [SPECIFIC ...]
                            If only loading, define which ontology/ies, projects
                            you want to load
    -n N_JOBS, --n_jobs N_JOBS
                            define number of cores used when importing data
    -w DOWNLOAD, --download DOWNLOAD
                            define whether or not to download imported data
    -u USER, --user USER  Specify a user name to keep track of who is building
                            the database
```

For a full update:

```
$ python builder.py --build_type full --download True --user ckg_user
```

3) Admin page (CKG app)

In CKG's app, the *Admin page* provides as well the option to start a full update of the database.



Warning: This option takes longer than the command-line options because multi-processing is not possible.

Partial update

If you want to update a specific ontology, database or project, you can use the partial update functionality in graphdb_builder. The partial update is done programmatically only and you will need to define which ontologies or databases names, or project identifiers to update. The ontologies and databases that are available to import can be checked in the configuration files: ontologies_config.yml under *ontologies* and databases_config.yml under *databases*. The partial update is again a 2-step process, so you will need to import the ontologies/databases/projects and then load them into the graph.

For instance, to update the Disease Ontology:

```
$ python builder.py --build_type import --import_types ontologies --data␣
↪disease --download True --user ckg_user
$ python builder.py --build_type load --load_entities ontologies --specific␣
↪disease --user ckg_user
```

To update a database:

```
$ python builder.py --build_type import --import_types databases --data HMDB␣
↪--download True --user ckg_user
$ python builder.py --build_type load --load_entities metabolite --user ckg_
↪user
```

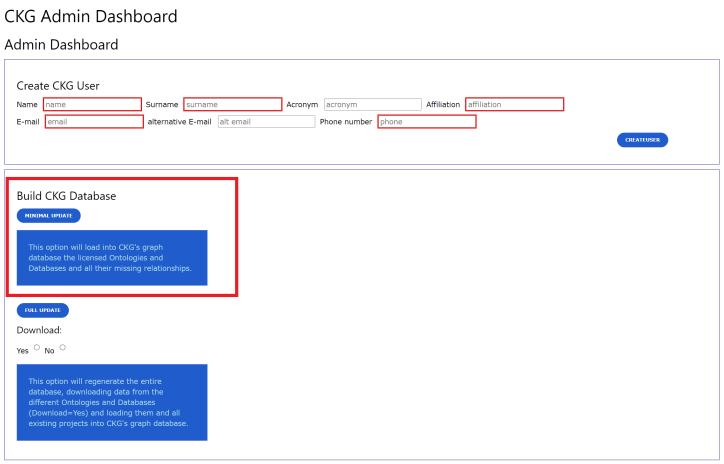
Minimal update

To easily have CKG up and running we provide a database dump that can be easily uploaded following *Building CKG's Graph Database from a Dump File*. However, as mentioned before licensed databases and ontologies are not provided in this dump file and you will need to apply for access to get them into CKG's graph database. When you get access to these data, you download them and store them into the right data folders (folders). Together with the dump file, you can download a folder with `data.tar.gz`, which contains relationships from existing nodes in the database to nodes created from the licensed databases (i.e. (:Drug)-[:HAS_SIDE_EFFECT]-(:Phenotype)). Unzip the contents in `CKG/data/` so that the `data` folder looks like the figure depicted.

Afterwards, you can run the minimal update, which will parse these databases and ontology and generate the missing nodes and relationships in the database.

```
$ cd CKG/ckg/graphdb_builder/builder
$ python builder.py -b minimal -u username
```

The Minimal update can be also done through the Admin page in CKG app.



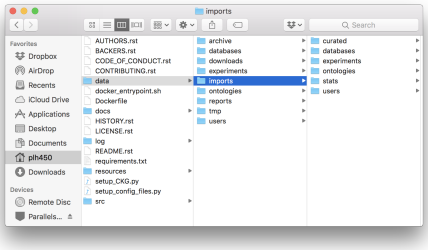


Fig. 4: Final CKG/data folder architecture.

Text mining update

The text mining results in CKG are collected from the Jensen Lab download page and are generated using the tagger tool. CKG collects the relevant *textmining_mentions files* (human, chemicals, diseases, tissues, etc) and parses them to obtain edgelists of protein-publication, drug-publication or disease-publication associations among other that can be loaded into the knowledge graph. These files are updated in a weekly basis, and given the number of open access publications made available in a week, it may be convinient to update the Knowledge Graph also with certain periodicity. For that, you can use the functionality `ckg_update_textmining` paired with a task scheduler to update the textmining results from the Jensen lab (https://jensenlab.org/).

To run the text mining update:

```
$ cd CKG
$ conda activate NAME_OF_YOUR_ENVIRONMENT
$ ckg_update_textmining
```

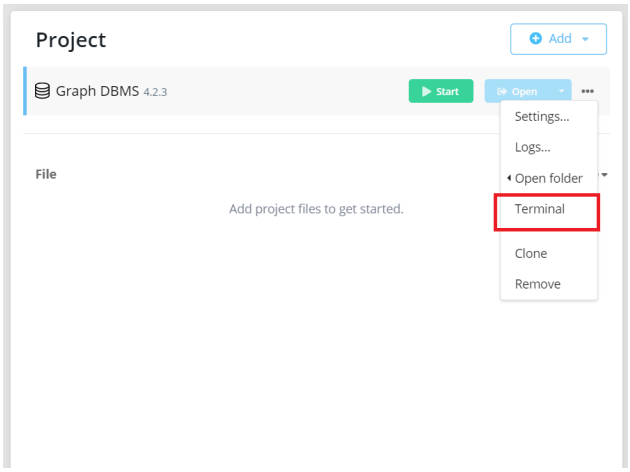
This will start a partial update of the Publication nodes and their relationships to other entities: tissues, diseases, proteins, etc. The update takes a while but can run in the background.

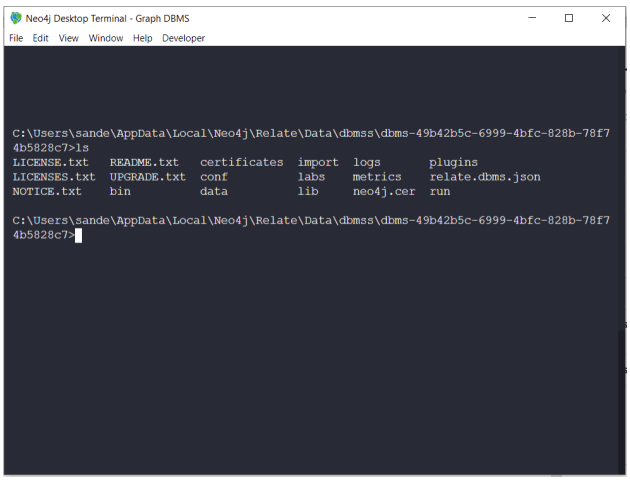
5.1.3 Building CKG's Graph Database from a Dump File

A dump file of the database is also made available in this link and alternatively, you can use it to load the graph database contained in it. To do so, download the dump file `ckg_latest_4.2.3.dump` and also `data.zip` from this `link <https://datashare.biochem.mpg.de/s/fP6MKhLRfceWwxC/download>, which contains some nodes and relationships associated to the licensed databases.

The `.dump` file will be used to load the Neo4j graph database:

If you have Neo4j desktop installed, you can use Neo4j terminal to access the path where the database is installed.





1. Create `backups` and `graph.db` folders:

```
$ mkdir backups
$ mkdir backups/graph.db
$ cp ckg_latest_4.2.3.dump backups/graph.db/.
```

1. After copying the dump file to backups/graph.db/, make sure the graph database is shutdown and run:

```
$ bin/neo4j-admin load --from=backups/graph.db/ckg_latest_4.2.3.dump --database=graph.
↪db --force
```

In some systems you might have to run this as root:

```
$ sudo bin/neo4j-admin load --from=backups/graph.db/ckg_080520.dump --database=graph.
↪db --force
$ sudo chown -R username data/databases/graph.db/
```

Warning: Make sure the dump file naming in the command above, matches the one provided to you.

Note: More information about restoring a database dump at Neo4j docs.

3. Once you are done, start the database and you will have a functional graph database.

More on the dump file

Another great use for the dump file, is to generate backups of the database (e.g. different versions of the imported biomedical databases). To generate a dump file of a specific Neo4j database, simply run:

```
$ cd /path/to/neo4jDatabases/database-identifier/installation-x.x.x/
$ bin/neo4j-admin dump --database=neo4j --to=backups/graph.db/name_of_the_file.dump
```

Warning: Remember to replace `name_of_the_file` with the name of the dump file you want to create.

5.2 Adding New Resources

The Clinical Knowledge Graph can be easily extended to include new ontologies, databases or experimental data types. The Graph Database Builder module (*graphdb_builder.py*) was built to quickly adapt the graph database data model to incorporate new nodes and relationships. For that, the data integrated from a new resouce into the graph is generated from parsers.

A **parser** is a script that interprets the content provided by the new resource, and translates it into a format that CKG can load into the database (nodes and relationships). In CKG node files define a table with a column named generally *ID* with a unique identifier for the node (i.e Gene Ontology identifier) and the rest of the columns are attributes of the node (i.e name or description). The relationship files define edge lists, that is, data structure used to represent a graph as a list of its edges. In these files, one column identifies one node (i.e START_ID) and another the node connected to it (i.e END_ID). The rest of the columns define attributes of the relationship (i.e score or type).

Node file example

"ID"	":LABEL"	"name"	"description"	"type"	"synonyms"
"GO:0000001"	"Gene_ontology"	"mitochondrion inheritance"	"The distribution of mitochondria, including the mitochondrial genome, into daughter cells after mitosis or meiosis, mediated by interactions between mitochondria and the cytoskeleton. [GOC:mcc, PMID:10873824, PMID:11389764]"	"-21"	"mitochondrion inheritance,mitochondrial inheritance"
"GO:0000002"	"Gene_ontology"	"mitochondrial genome maintenance"	"The maintenance of the structure and integrity of the mitochondrial genome; includes replication and segregation of the mitochondrial chromosome. [GOC:ai, GOC:vw]"	"-21"	"mitochondrial genome maintenance"
"GO:0000003"	"Gene_ontology"	"reproduction"	"The production of new individuals that contain some portion of genetic material inherited from one or more parent organisms. [GOC:go_curators, GOC:isa_complete, GOC:jl, ISBN:0198506732]"	"-21"	"reproduction,reproductive physiological process,Wikipedia:Reproduction"
"GO:0000011"	"Gene_ontology"	"vacuole inheritance"	"The distribution of vacuoles into daughter cells after mitosis or meiosis, mediated by interactions between vacuoles and the cytoskeleton. [GOC:mcc, PMID:10873824, PMID:14616069]"	"-21"	"vacuole inheritance"
"GO:0000012"	"Gene_ontology"	"single strand break repair"	"The repair of single strand breaks in DNA. Repair of such breaks is mediated by the same enzyme systems as are used in base excision repair. [PMID:18626472]"	"-21"	"single strand break repair"

Relationship file example

"START_ID"	"END_ID"	"TYPE"	"description"	"type"	"synonyms"
"GO:0072116"	"GO:0048645"	"HAS_PARENT"	"The distribution of mitochondria, including the mitochondrial genome, into daughter cells after mitosis or meiosis, mediated by interactions between mitochondria and the cytoskeleton. [GOC:mcc, PMID:10873824, PMID:11389764]"	"-21"	"mitochondrion inheritance,mitochondrial inheritance"
"GO:0046280"	"GO:0046275"	"HAS_PARENT"	"The maintenance of the structure and integrity of the mitochondrial genome; includes replication and segregation of the mitochondrial chromosome. [GOC:ai, GOC:vw]"	"-21"	"mitochondrial genome maintenance"
"GO:0009238"	"GO:0009712"	"HAS_PARENT"	"The production of new individuals that contain some portion of genetic material inherited from one or more parent organisms. [GOC:go_curators, GOC:isa_complete, GOC:jl, ISBN:0198506732]"	"-21"	"reproduction,reproductive physiological process,Wikipedia:Reproduction"
"GO:0032190"	"GO:0003211"	"HAS_PARENT"	"The distribution of vacuoles into daughter cells after mitosis or meiosis, mediated by interactions between vacuoles and the cytoskeleton. [GOC:mcc, PMID:10873824, PMID:14616069]"	"-21"	"vacuole inheritance"
"GO:0071386"	"GO:0051384"	"HAS_PARENT"	"The repair of single strand breaks in DNA. Repair of such breaks is mediated by the same enzyme systems as are used in base excision repair. [PMID:18626472]"	"-21"	"single strand break repair"

The files and format provided by the different resources are not always as standard or stable as we would like (*see note*). For that reason, CKG uses **configuration** files to define things like: resource's url, columns to be extracted, files to be generated and columns to be stored. For example, the configuration file for the STRING and STITCH databases is:

```
###### STRING database #######
STRING_cutoff: 0.4

STRING_mapping_url: 'https://stringdb-static.org/download/protein.aliases.v11.0/9606.
↪protein.aliases.v11.0.txt.gz'
STITCH_mapping_url: 'http://stitch.embl.de/download/chemical.aliases.v5.0.tsv.gz'

STITCH_url: 'http://stitch.embl.de/download/protein_chemical.links.detailed.v5.0/9606.
↪protein_chemical.links.detailed.v5.0.tsv.gz'
STRING_url: 'https://stringdb-static.org/download/protein.links.detailed.v11.0/9606.
↪protein.links.detailed.v11.0.txt.gz'

STRING_actions_url: 'https://stringdb-static.org/download/protein.actions.v11.0/9606.
↪protein.actions.v11.0.txt.gz'
STITCH_actions_url: 'http://stitch.embl.de/download/actions.v5.0/9606.actions.v5.0.
↪tsv.gz'

header:
    - 'START_ID'
    - 'END_ID'
    - 'TYPE'
    - 'interaction_type'
    - 'source'
    - 'evidence'
    - 'scores'
    - 'score'

header_actions:
    - 'START_ID'
    - 'END_ID'
    - 'TYPE'
    - 'action'
    - 'directionality'
    - 'score'
    - 'source'
```

Here, we specify the urls where to get the data from, and the headers in the resulting tsv files (relationships: interacts_with, acts_on).

When you want to add a new resouce, we recommend you to define first how this will impact the existing data model. For instance, try to answer the following questions:

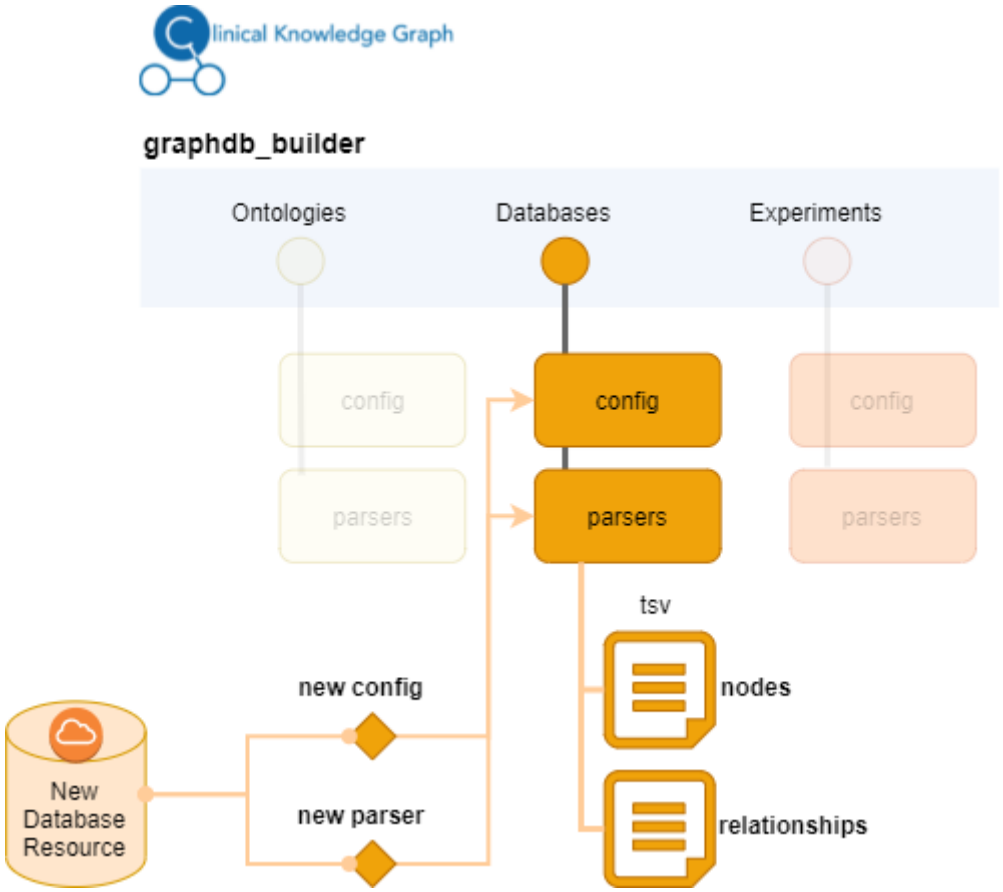
- Are you adding new information or complementing existing resources? if complemnenting, what's the overlap between the different sources?
- What types of clinical/biological questions would you like to answer with the new data?
- Will the update create new node types? What's the preferred ontology/terminology for such node type?
- How will the new data be integrated in the existing data model (connectivity)?
- What relationship types will you integrate? What attributes could be useful/relevant?

Note: To simplify the maintenance of parsers, we created unit tests to validate the existing urls for the resources integrated in CKG (graphdb_builder test).

5.2.1 Adding a New Resource: Ontology or Database

To add a new database, you will first need to identify how to obtain the data. If available, we recommend downloading the data in text format directly from the resource. In the case of Ontologies, use the Open Biological and Biomedical Ontologies format. Once idenitified the url where to obtain the data, you can start defining the **configuration** file. Then, you will need to understand the format of the data, what information can be retrieved and whether it can be associated to the new nodes or to the relationships created. The next step will be to define how the resulting tsv files should look like. Implement the parser according to this by extracting the data from the original format and translate them into what the output files should look like. We recommend use as template some of the existing parsers. Whenever possible use the **configuration** file to define variables that may need to be changed at some point such as for instance ulrs, cutoffs, lists of columns, etc.

Include the **configuration** and the new **parser** files into the right graphdb_builder directory (ontologies, databases). Ontlogies work in slightly different way, because they generally follow a standard format (obo) all the configuration is defined in the general configuration file (ontologies_config.yml). **Databases will need a specific configuration file.**



Example: Adding a new database resource

To make the new resource available:

1) Include the new resource in the list of ontologies or databases defined in the general configuration files, ontologies_config.yml or databases_config.yml respectively.

2) If the new resource is a database, include in database_controller.py a new conditional block with the name of the new resource (same as in 1)).

   For instance, if new nodes and relationships are created, we woud add something like this:

```
elif database.lower() == "NAME_NEW_RESOURCE":
    name_new_resourceParser.parser(database_directory, importDirectory,␣
↪download, updated_on)
```

This piece of code will call the parser which will write the new nodes and relationships directly in the importDirectory.

3) Add new queries in the cypher.yml file to load the nodes and relationships from the generated files into the graph. The identifier used here, will be used when loading the new entity to find the right queries (i.e. IMPORT_NEW_ENTITY).

4) If new nodes and/or relationships are created, add the new entity to the graph list in the builder_config.yml file. If we are adding a complementary resource providing existing relationship types, we will need to add the new resource to the list of existing resources. For instance, this is how we integrated multiple pathway resources ('Reactome', 'SMPDB'). If that is the case, remember to include the name of the resource in the name of the generated file (RESOURCE_relationship_type.tsv).

5) If new nodes and/or relationships are created, include a new condition block in loader.py file. For instance:

```
elif i == "NEW_ENTITY_NAME":
    code = cypher_queries['IMPORT_NEW_ENTITY']['query']
    queries = code.replace("IMPORTDIR", import_dir).split(';')[0:-1]
```

5.2.2 Adding a New Experimental Data Type

CKG can also be easily extended to integrate new experimental data types. It works in a similar way to what we describe in the previous section. Each data type has its own **configuration** and **parser**.

In this case, the configuration file defines which columns from the input format are used and which ones contain information to build relationships specific for each analytical sample (i.e. quantification for each sample). Depending on the data type integrated, the input format can be diverse so the configuration can describe different input formats (processing tools) and how to use them. For example, check out the proteomics configuration file) where we define different configuration for the output of several different processing tools (MaxQuant, Spectronaut, DIA-NN, FragPipe, mzTab)

The parser is then used to translate the input format into the relevant relationships to be loaded into CKG. For instance, in the case of proteomics, the proteomics parser generates the relationship (:Analytical_sample)-[:HAS_QUATIFIED_PROTEIN]-(:Protein) and defines *value* as an attribute of the relationship to store the quantified intensity. In order to identify the value columns (Analytical_sample), we use a regular expression defined in the configuration file.

When the new parser is created, you will need to:

1) Add a condtional block in experimental_controller.py to include the new data type. As an example:

```
elif dataType == "NEW_DATA_TYPE":
    data = newDataTypeParser.parser(projectId, dataType)
    for dtype, ot in data:
        generate_graph_files(data[(dtype, ot)], dtype, projectId, stats, ot,␣
↪dataset_import_dir)
```

2) Add cypher queries to load the new data type into the graph. For that, inlclude the data type as part of the *IMPORT_DATASETS* structure in the cypher.yml file and specify the necessary cypher queries for the new data type.

Note: This will add the new data type into the knowledge graph, but if you want to generate reports for this data type,

you will need to include it as a new Dataset class, define how the data should be retrieved and also how they should be analyzed (see *CKG Project Report*).

5.2.3 Adding New Data Formats

Some Omics data types can be processed using different software that generate non-standard output files to be parsed. In CKG, you can use the data type **configuration** file to define how the information from these formats should be translated into nodes and relationships of the specific data type. This is the case in Proteomics data, and you can see how we use different configuration for each processing tool proteomics configuration file) and the same parser for all of them (proteomics parser).

CHAPTER

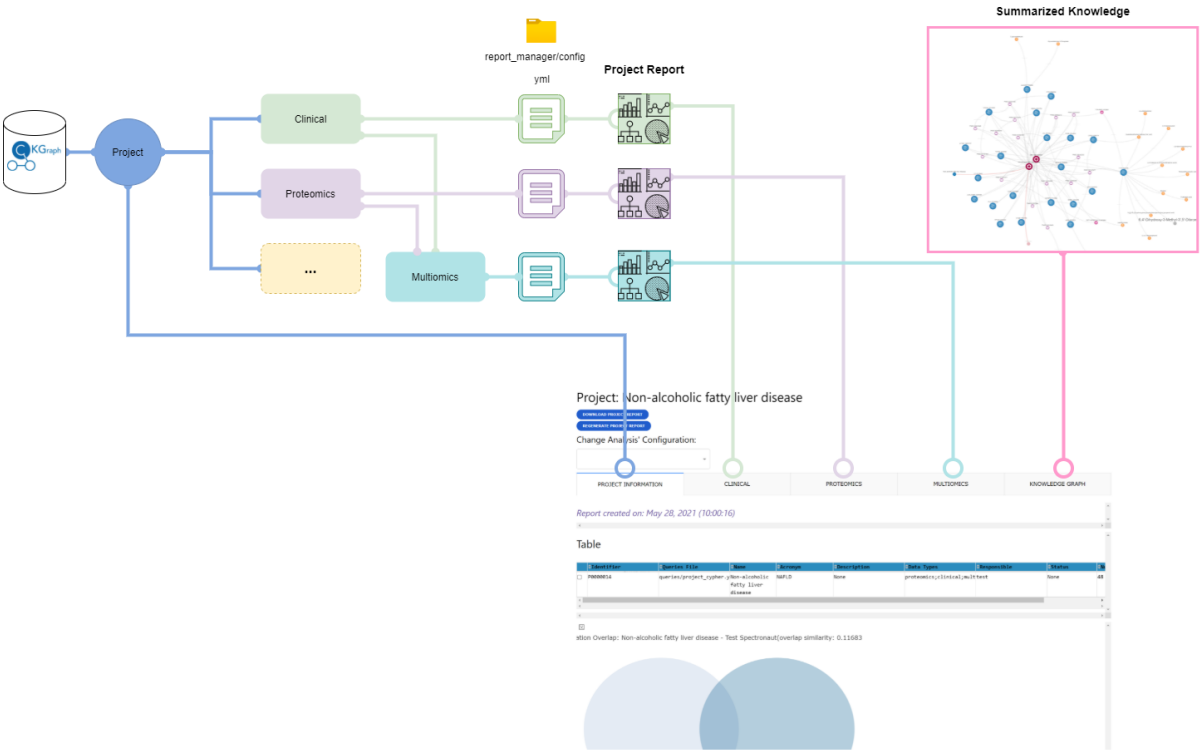
SIX

PROJECT REPORT

- **Generate a project**: *Project*

6.1 CKG Project Report

CKG can easily generate automated statistical reports using data linked to existing projects in the graph database. These reports can be generated once the experimental data are uploaded.



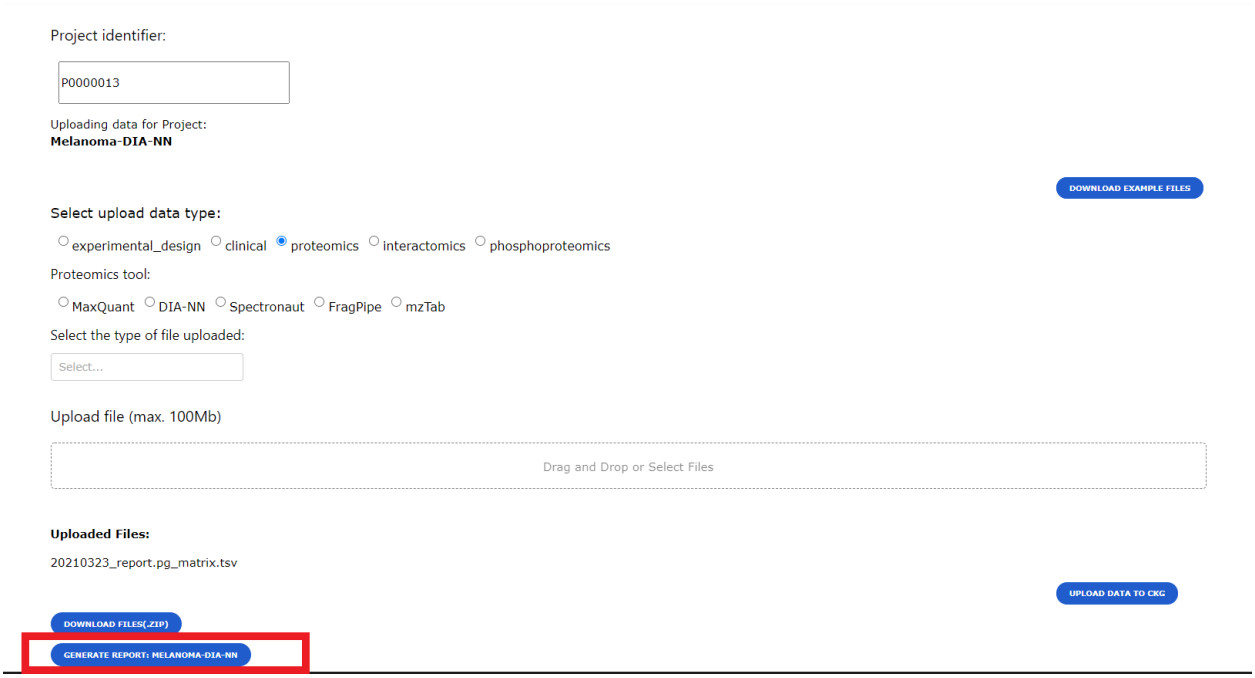
The sequence of analyses shown in the report is based on configuration files associated to each data type available. CKG has default configuration files for the currently integrated data types - clinical, proteomics, phosphoproteomics and interactomics datasets.

Users can build customized analytical pipelines by defining their own configuration files. Details on how to build these configuration files can be found in section *Define data analysis parameters*.

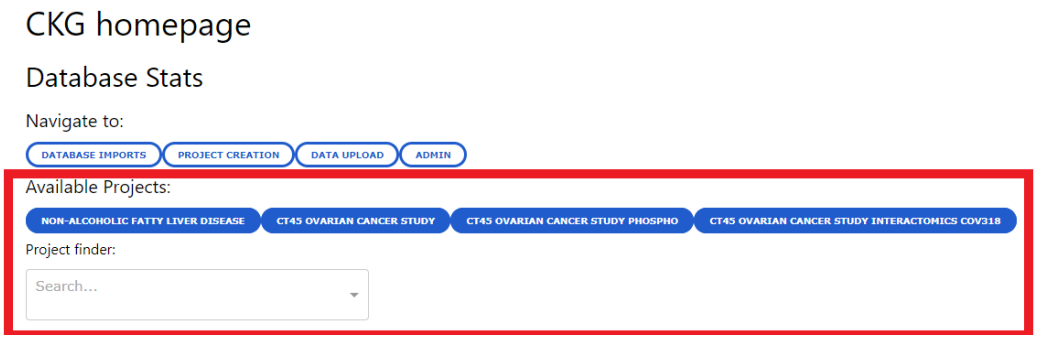
When a Report is generated, users will be able to access this report either through CKG app or programmatically for instance through a Jupyter notebook (see Access Project Report notebook).

6.1.1 CKG app Report

To generate a report for a specific project all the data for the project needs to be uploaded into the Knowledge Graph database. When uploading data into CKG through the Data Upload page (see Upload Data), users will be given the option to generate the report.

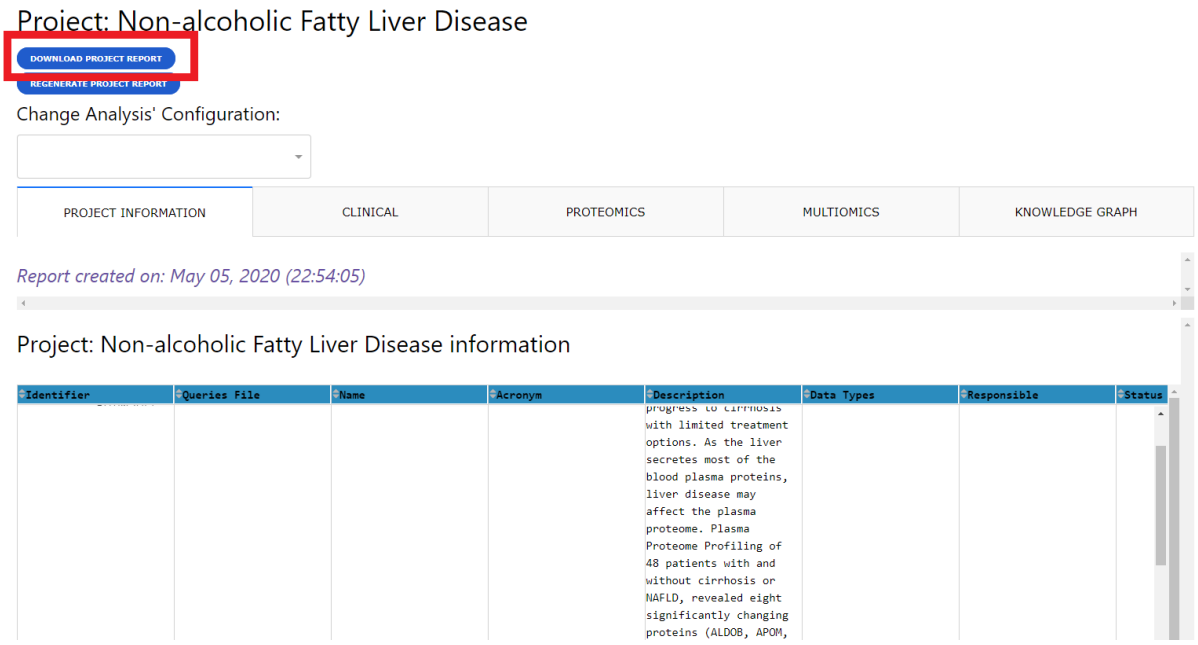


Users will be able to generate reports for the available projects or have access to the already generated reports from the Home page where they will see links to several projects in the database or they will be able to search for a specific project in a dropdown menu and by selecting a project, a button to generate/access the project report will be created.



When accessing a report, if the report was not previously generated CKG will run all the default analyses defined in the configuration files (*Define data analysis parameters*). If the report was already generated, CKG will just load the latest report and show the date when it was created.

Reports can be downloaded using the download button. This will generate a compressed folder (zip) with all the analysis and visualization generated by CKG. Also, the download will include the configuration files used to generate those results, which can be reused to reproduce or replicate the project.



Note: Network visualizations are stored in 3 different formats: json, node and edge tables and GML (compatible with Cytoscape).

Report structure

A project report is divided into multiple tabs with one with some project information and other with analysis results for each dataset available (i.e. clinical, proteomics).

Project Info Tab

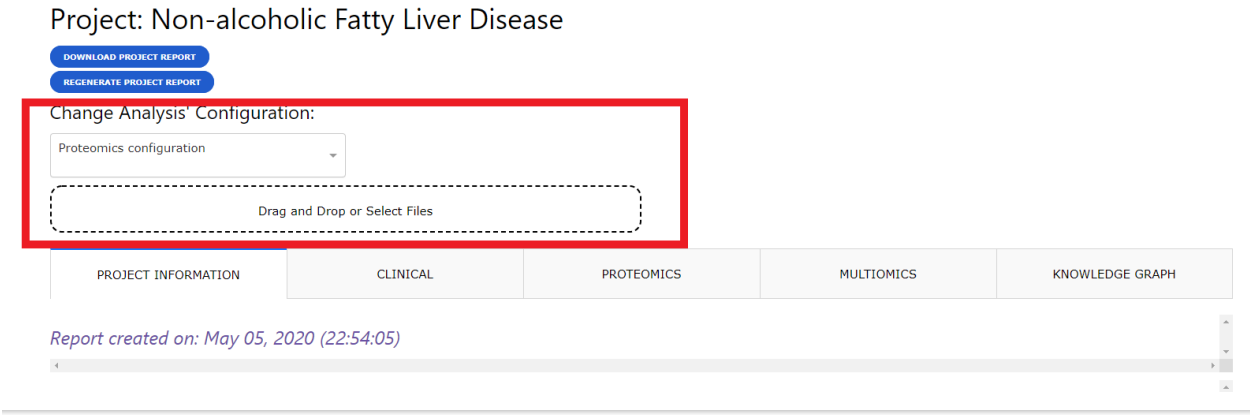
The project info tab contains details about the project that were specified upon creation and a similarity analysis comparing it with other existing projects in the Knowledge Graph. This comparison is at the proteomics level and measured with 2 distinct similarity scores:

- **Correlation**: Pairwise Pearson correlation coefficient (cutoff=0.5)
- **Overlap**: Jaccard index

Further, for the projects found most similar (correlation), CKG will provide a network showing if these projects are somehow connected, for instance, if they study the same disease or the same tissue.

Dataset Analysis Tabs

There will be a tab for each of the data types associated with a project. In these tabs you will see the results from the analysis specified in the configuration files. Initially these tabs will present the results defined in the default analytical pipelines but users can regenerate the reports by providing customized configuration files for one or several data types.

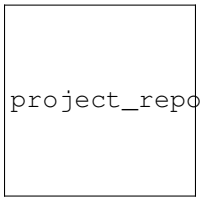


Multiomics Tab

In the previous section we described how dataset independent analyses are shown in the report. However, CKG also analyzes different data types combined and shows the results in the Multiomics tab. Currently, CKG can analyze clinical and proteomics data together in a multi-correlation network analysis or using Weighted Gene Co-expression Network Analysis (WGCNA). These analysis need to be defined in a configuration file as well (*Multiomics data analysis parameters*).

Knowledge Graph Tab

CKG will add another tab to the report where it will try to summarize all the resul ts obtained in the independent and multiomics analysis. CKG will connect relevant hits (clinical variables, proteins, etc) in the analyses and connect them to knowledge in the graph database to build a project-specific knowledge graph (diseases, drugs, protein complexes, etc.). Then, in order to summarize the results to make them easier to interpret, CKG uses betweenness centrality to identify the most relevant nodes in the generated network. The summarization is visualized as network or a Sankey plot.



CKG offers more summarization options based also on the concept of centrality but also other algorithms such as PageRank. For examples of how to use this summarization algorithms check the notebook recipes: Annotate Proteins with CKG Knowledge and Annotate drugs with CKG Knowledge.

CHAPTER

SEVEN

NOTEBOOKS

- **Jupyter Notebooks**: *Notebooks*

7.1 The Clinical Knowledge Graph Notebooks

The Jupyter Notebook is used to interact with the notebooks provided in the Clinical Knowledge Graph. This open-source application allows you to create and share code, visualise outputs and integrated multiple big data tools.

In order to get started, make sure you have Python installed (3.3 or greater) as well as Jupyter Notebook. The latter can be installed using pip (see below).For more detailed instructions, visit the official guide.

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install jupyter
```

Congratulations! Now you can run the notebook, by typing the following command in the Terminal (Mac/Linux):

```
$ jupyter notebook
```

Or,

```
$ jupyter-notebook
```

As part of the Clinical Knowledge Graph package, we provide a series of Jupyter notebooks to facilitate the analysis of data, database querying and the use of multiple visualisation tools. These notebooks can be found in `ckg/notebooks`, under `reporting` or `development`.

Note: If you would like to use two instances of the same notebook, just duplicate in-place and modify the name accordingly.

Warning: If the Clinical Knowledge Graph is deployed in a server, please set up a Jupyter Hub in order to allow access to the Jupyter Notebook.

7.1.1 Recipes notebooks

In `ckg/notebooks/recipes` we gathered Jupyter notebooks with analysis and workflows we believe are of interest for the users but are still under development.

- **Access Project Report**

Easy access to all the projects in the graph database. Loads all the data from a specific project (e.g. "P0000001") and shows the report in the notebook. This notebook enables the visualisation of all the plots and tables that constitute a report, as well as all the dataframes used and produced during its generation. By accessing the data directly, you can use the python functionality to further the analysis or visualisation.

- **Working with R**

Notebook entirely written in R. One of the many advantages of using Jupyter notebooks is the possibility of writing in different programming languages. In this notebook, we demonstrate how R can be used to, similarly to *project_reporting.ipynb*, load a project and explore the analysis and plots.

In the beginning of the notebook, we create custom functions to load a project, read the report, read a dataset, and plot and network from a json file. Other R functions like these can be developed by the users according to their needs.

- **Parallel plots**

An example of a new interactive visualisation method, not currently implemented in the Clinical Knowledge Graph, but using data from a stored project. We start by loadingall the data and the report of a specific project (e.g. "P0000001"), and accessing different dataframes within the proteomics dataset, as well as, the correlation network.This plot is then converted into a Pandas DataFrame and used as input for the interactive Parallel plot.

The function is created and made interactive with Jupyter Widgets `interact` function (`ipywidgets.interact`), which automatically creates user interface (UI) controlswithin the notebook. In this case, the user can select different clusters of proteins (from the correlation network) and observe their variation across groups.

- **Download PRIDE data**

Easily download data directly from PRIDE (https://www.ebi.ac.uk/pride/) by specifying the PRIDE identifier and the file name to download. The notebook also shows how toformat the data and analyze them with CKG.

- **Power Analysis**

Power anlysis based on an existing project in CKG. It allows to define the sample size needed to achieve a specific statistical power using effect sizes fromprevious analyses.

- **single sample Gene Set Enrichment Analysis**

Perform single sample Gene Set Enrichment Analysis (ssGSEA) using different gene/protein sets and visualize them using Principal Component Analysis (PCA).

- **Annotate Proteins with CKG Knowledge**

This notebook shows how to extract knowledge associated with a list of proteins and summarize it using different methods: centrality, pagerank.

- **Annotate drugs with CKG Knowledge**

This notebook shows how to extract knowledge associated with a list of drugs and summarize it using different methods: centrality, pagerank.

- **Batch effect correction**

This notebook exemplifies how to correct batch effects in a proteomics experiment and how to add this correction in CKG's analytical pipeline (configuration).

- **Convert SDRF format to CKG**

This notebook shows CKG's functionality to convert SDRF format into CKG and viceversa.

- **Upload a SDRF file to CKG**

How to upload programmatically a SDRF file into CKG.

- **Convert MzTab format to CKG**

This notebook shows CKG's functionality to read MzTab format and convert it into CKG proteomics/metabolomics format (edge lists).

7.1.2 Reports notebooks

Reports notebooks

- **Plasma proteome profiling discovers novel proteins associated with non-alcoholic fatty liver disease**

A Jupyter notebook reanalyzing the study Plasma proteome profiling discovers novel proteins associated with non-alcoholic fatty liver disease (https://www.embopress.org/doi/full/10.15252/msb.20188793). The analyses shown in the notebook reproduce CKG's default analytical pipeline from data processing to knowledge.

- **Urachal Carcinoma Case Study**

Jupyter notebook depicting the use of the Clinical Knowledge Graph database and the analytics core as a decision support tool, proposing a drug candidate in a specific subject case.

The project is analysed with the standard analytics workflow and a list of significantly differentially expressed proteins is returned. To further this analysis, we first filterfor regulated proteins that have been associated to the disease in study (lung cancer); we then search the database for known inhibitory drugs for these proteins; and to narrow downthe list, we can query the database for each drug's side effects. The treatment regimens are also available in the database and their side effects can be used to rank the proposed drugs.We can prioritise drugs with side effects dissimilar to the ones that caused an adverse reaction in the patient, and identify papers where these drugs have already been associated to thestudied disease, further reducing the list of potential drugs candidates.

- **Proteomics-Based Comparative Mapping of the Secretomes of Human Brown and White Adipocytes**

A Jupyter notebook that exemplifies how to analyse any external dataset deposited in EBI's PRIDE database, by simply using the respective PRIDE identifier for the project (PXD. . .)and CKG's Analytics Core.

The entire project folder is downloaded from PRIDE, decompressed, and the proteinGroups.txt file is parsed to obtain relevant information including LFQ intensities. After being converted intoa wide-format dataframe, the default analysis pipeline implemented in the CKG, is reproduced by calling the respective functions directly from the Analytics Core modules(*analytics_core.analytics* and *analytics_core.viz*).

Additionally, we demonstrate how other publicly accessible biomedical databases can be downloaded into the notebook, and their information mined for relevant metadata. In this specific case, the Human Protein Atlas is downloaded and mined in order to filter for known or predicted secreted proteins.

- **Covid-19 Olink Analysis from Massachusetts General Hospital**

A Jupyter notebook showing how CKG can be used to analyze a dataset generated with a different technology, in this case Olink.

The analysis reproduces similar results to the ones described in this manuscript: https://www.biorxiv.org/content/10.1101/2020.11.02.365536v2 and also extends these results by comparing different severity groups in this cohort based on WHO scores.

- **Meduloblastoma Data Proteomics Re-analysis**
- **Meduloblastoma Data Analysis-SNF**

These two Jupyter notebooks show:

1) A re-analysis of this study: Proteomics, Post-translational Modifications, and Integrative Analyses Reveal Molecular Heterogeneity within Medulloblastoma Subgroups (https://www.sciencedirect.com/science/article/pii/S1535610818303581). This analysis uses the proteomics data to compare the clinical Subgroups
2) A multiomics approach integrating all the molecular data available: proteomics, RNA sequencing and PTMs. CKG uses Similarity Network Fusion (https://www.nature.com/articles/nmeth.2810) to integrate these datasets and to define Medulloblastoma subgroups. The results are further explained in this presentation_meduloblastoma.

- **CPTAC Glioblastoma Discovery Study Proteomics Re-analysis**

A re-analysis of the CPTAC Discovery Study using the Proteomics data of 100 brain tumor samples and 10 normal samples (https://cptac-data-portal.georgetown.edu/study-summary/S048). More details on the results available in this presentation_gbm.

CHAPTER

EIGHT

MORE FEATURES

- **CKG Import Statistics**: *Imports stats*
- **Retrieving data from the CKG**: *DB Querying*

8.1 Clinical Knowledge Graph Statistics: Imports

Everytime you perform a *Full update* or *Partial update* of any of the ontologies or databases, CKG will register the number of nodes and relationships imported (parsed into tsv files). When importing ontologies or databases, each parser will return `stats`, which is a Pandas dataframe with the total number of nodes and relationships extracted from the ontology or database source and details about the time when the imoport happened or the disk space occupied by the generated file.

The columns in the `stats` dataframe are:

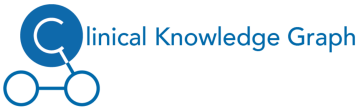
- date: import date
- time: import time
- dataset: name ontology or database
- filename: name of the file containing the nodes or relationships
- file_size: size of the file
- Imported_number: number of nodes or relationships
- Import_type: nodes (entity) or relationships
- name: node or relationship type
- updated_on: date when it was downloaded
- import_id: unique id import

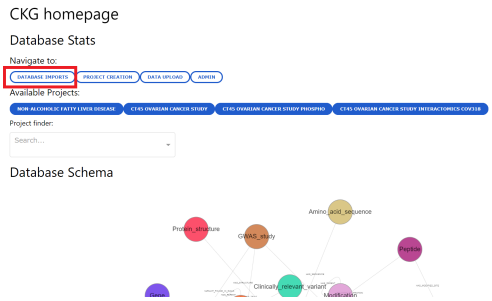
This dataframe is stored in the *data/stats* directory as a Hierarchical Data Format file (HDF5) and can be read directly from pandas:

```
import pandas as pd

stats_file = 'data/stats/stats.hdf'

df = pd.read_hdf(stats_file)

df.head()
```

date	time	dataset	filename	file_size	Imported_number	Import_type	name	updated_on	import_id
2021-5-26	8:46:8	drug-bank	databasesDrug.tsv	1452455	14315	entity	Drug	2021-05-26	dc0cd359-e8db-4776-b881-42ef92b5a374
2021-5-26	8:46:16	drug-bank	databasesdrug-bank_interacts_with_drug.tsv	37445782	2682157	relationships	drug-bank_interacts_with_drug	2021-05-26	dc0cd359-e8db-4776-b881-42ef92b5a374
2021-5-26	8:46:8	drug-bank	databasesdrug-bank_annotated_in_pathway.tsv	185248	3780	relationships	drug-bank_annotated_in_pathway	2021-05-26	dc0cd359-e8db-4776-b881-42ef92b5a374
2021-5-26	8:46:8	drug-bank	databasesdrug-bank_targets_protein.tsv	595700	14890	relationships	drug-bank_targets_protein	2021-05-26	dc0cd359-e8db-4776-b881-42ef92b5a374

Also, you can visualize these statistics in different plots as part of CKG app, just navigate to *Data imports*:





advanced_features/../_static/images/database_imports_app.png

In this web app, you will access a report with statistics such as:

- Number of imported nodes (entities) and relationships by date/time
- Number of imported nodes (entities) and relationships by date/time per ontology/database
- File sizes for nodes and relationships
- All stats for each node type and relationship per database

These statistics are quite relevant to follow the progress of the knowledge graph and also a good way to compare the current import of an ontology or a database with respect to previous updates.

Note: We recommend monitoring these statistics to identify possible issues when updating CKG.

8.2 Retrieving data from the Clinical Knowledge Graph database

CKG has multiple cypher queries predefined to extract knowledge from the graph. These queries are part of:

- Graph Database Builder queries: define how to load data into the graph (see graphdb_builder cypher.yml)
- Report manager queries: - Dataset queries: extract data and knowledge for each data type integrated and analyzed in the graph for analysis with the analytics core (dataset_cypher.yml) - Knowledge queries:
  - Annotations: extract knowledge based on list of proteins or drugs (knowledge_annotation.yml)

These queries have been defined in YAML format. This structure allows assigning attributes such as name, description or involved nodes and relationships, which make the queries searchable using the functionality we implemented in query_utils.py.

For instance, we can use functions in this module to find within a query file, which queries involve specific types of nodes or relationships:

```
selected_queries = {}
queries = query_utils.read_queries(queries_file="ckg/report_manager/queries/knowledge_
↪annotation.yml")
for data_type in queries:
    selected_queries[data_type] = query_utils.find_queries_involving_
↪nodes(queries=queries[data_type], nodes=["Protein", "Disease"], print_pretty=True)
```

If you want to contribute, you can add new queries following the same structure and they will then be available for everyone.

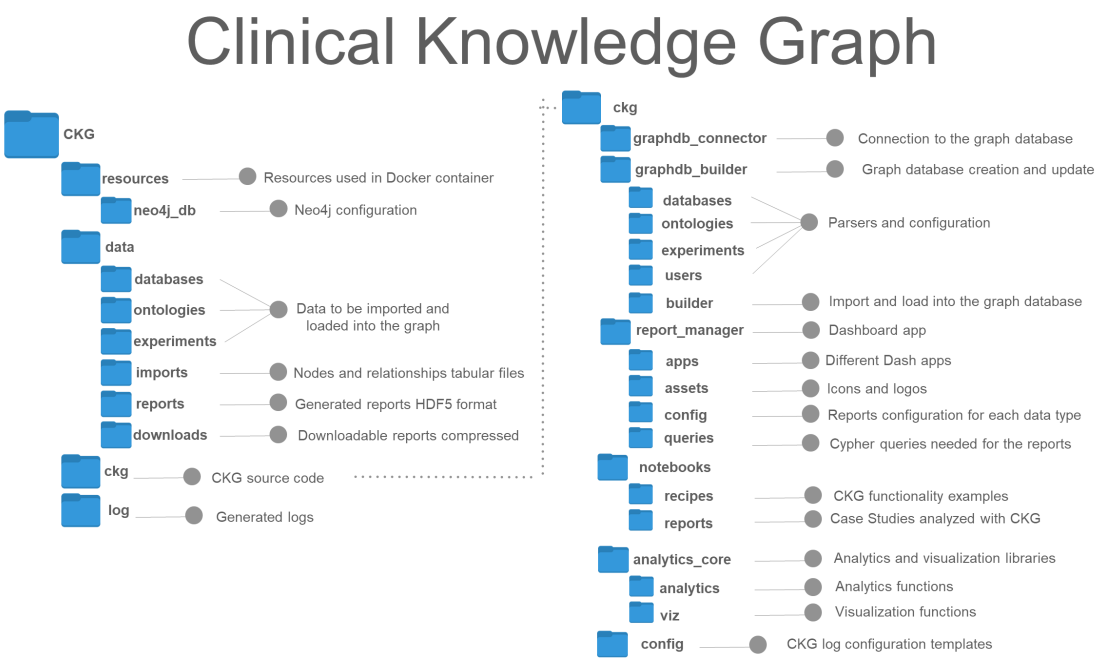
```
identifier:
    name: ... # string
    description: ... # string
    involved_nodes: # list
      - ... # node type in the graph
      - ...
    involved_rels: # list
      - ... # relationship type in the graph
    query: > # string
        ...
        ...
        ... # Cypher query
```

CHAPTER

NINE

API REFERENCE

Code available in GitHub.



9.1 API Reference

9.1.1 Graph Database Connector (graphdb_connector)

connector.py

read_config()

getGraphDatabaseConnectionConfiguration(*configuration=None*, *database=None*)

connectToDB(*host='localhost'*, *port=7687*, *user='neo4j'*, *password='password'*)

removeRelationshipDB(*entity1*, *entity2*, *relationship*)

modifyEntityProperty(*parameters*)

parameters: tuple with entity name, entity id, property name to modify, and value

do_cypher_tx (*tx*, *cypher*, *parameters*)

commitQuery (*driver*, *query*, *parameters={}*)

sendQuery (*driver*, *query*, *parameters={}*)

getCursorData (*driver*, *query*, *parameters={}*)

find_node (*driver*, *node_type*, *parameters={}*)

find_nodes (*driver*, *node_type*, *parameters={}*)

run_query (*query*, *parameters={}*)

generate_virtual_graph (*graph_json*)

query_utils.py

read_knowledge_queries (*dataset_type='proteomics'*)

read_queries (*queries_file*)

list_queries (*queries*)

find_queries_involving_nodes (*queries*, *nodes*, *print_pretty=False*)

find_queries_involving_relationships (*queries*, *rels*)

get_query (*queries*, *query_id*)

get_description (*query*)

get_nodes (*query*)

get_relationships (*query*)

map_node_name_to_id (*driver*, *node*, *value*)

9.1.2 Graph Database Builder (graphdb_builder)

Ontology Databases

Ontologies Parsers

efoParser.py

parser (*ontology_files*)

icdParser.py

parser (*ICDfile*)

Parses and extracts relevant data from ICD-10 files (Classification of Diseases).

Parameters **ICDfile** – list of files downloaded from the ontology database and used to generate nodes and relationships to the graph database.

Returns

Three nested dictionaries: terms, relationships between terms, and definitions of the terms.

- terms: Dictionary where each key is an ontology identifier (*str*) and the values are lists of names and synonyms (*list[str]*).
- relationships: Dictionary of tuples (*str*). Each tuple contains two ontology identifiers (source and target) and the relationship type between them.
- definitions: Dictionary with ontology identifiers as keys (*str*), and definition of the terms as values (*str*).

oboParser.py

parser(*ontology*, *files*)

Multiple ontology database parser. This function parses and extracts relevant data from: Disease Ontology, Tissues, Human Phenotype Ontology, HUPO-PSI and Gene Ontology databases.

Parameters

- **ontology** (*str*) – name of the ontology to be imported ('Disease', 'Tissue', 'Phenotype', 'Experiment', 'Modification', 'Molecular_interactions', 'Gene_ontology')
- **files** (*list*) – list of files downloaded from an ontology and used to generate nodes and relationships in the graph database.

Returns

Three nested dictionaries: terms, relationships between terms, and definitions of the terms.

- terms: Dictionary where each key is an ontology identifier (*str*) and the values are lists of names and synonyms (*list[str]*).
- relationships: Dictionary of tuples (*str*). Each tuple contains two ontology identifiers (source and target) and the relationship type between them.
- definitions: Dictionary with ontology identifiers as keys (*str*), and definition of the terms as values (*str*).

reflectParser.py

parser(*files*, *filters*, *qtype=None*)

Parses and extracts relevant data from REFLECT ontologies: Disease Ontology, Tissues, STITCH and Gene Ontology databases.

Parameters

- **files** (*list*) – list of files downloaded from an ontology and used to generate nodes and relationships in the graph database.
- **filters** (*list*) – list of ontology identifiers to be ignored.
- **qtype** (*int*) – ontology type code.

Returns

Three nested dictionaries: terms, relationships between terms, and definitions of the terms.

- terms: Dictionary where each key is an ontology dentifier (*str*) and the values are lists of names and synonyms (*list[str]*).
- relationships: Dictionary of tuples (*str*). Each tuple contains two ontology identifiers (source and target) and the relationship type between them.

- definitions: Dictionary with ontology dentifiers as keys (*str*), and definition of the terms as values (*str*).

snomedParser.py

parser(*files*, *filters*)

Parses and extracts relevant data from SNOMED CT database files.

Parameters

- **files** (*list*) – list of files downloaded from SNOMED CT and used to generate nodes and relationships in the graph database.
- **filters** (*list*) – list of SNOMED CT Identifiers to be ignored.

Returns

Three nested dictionaries: terms, relationships between terms, and definitions of the terms.

- terms: Dictionary where each key is a SNOMED CT Identifier (*str*) and the values are lists of names and synonyms (*list[str]*).
- relationships: Dictionary of tuples (*str*). Each tuple contains two SNOMED CT Identifiers (source and target) and the relationship type between them.
- definitions: Dictionary with SNOMED CT Identifiers as keys (*str*), and definition of the terms as values (*str*).

get_inactive_terms(*concept_file*)

Parameters **concept_file** –

Return set inactive_terms inactive terms

ontologies_controller.py

Biomedical Databases

Biomedical Databases Parsers

cancerGenomeInterpreterParser.py

corumParser.py

parser(*databases_directory*, *download=True*)

disgenetParser.py

parser(*databases_directory*, *download=True*)

readDisGeNetProteinMapping(*config*, *directory*)

readDisGeNetDiseaseMapping(*config*, *directory*)

drugBankParser.py

drugGeneInteractionDBParser.py

exposomeParser.py

foodbParser.py

goaParser.py

gwasCatalogParser.py

parser(*databases_directory*, *download=True*)

hgncParser.py

parser(*databases_directory*, *download=True*)

hmdbParser.py

hpaParser.py

intactParser.py

parser(*databases_directory*, *download=True*)

jensenlabParser.py

mutationDsParser.py

parser(*databases_directory*, *download=True*)

oncokbParser.py

pathwayCommonsParser.py

parser(*databases_directory*, *download=True*)

pfamParser.py

pspParser.py

reactomeParser.py

refseqParser.py

parser(*databases_directory*, *download=True*)

siderParser.py

signorParser.py

parser(*databases_directory*, *download=True*)

parse_substrates(*filename*, *modifications*, *accronyms*, *amino_acids*)

smpdbParser.py

parser(*databases_directory*, *download=True*)

parsePathways(*config*, *fhandler*)

parsePathwayProteinRelationships(*fhandler*)

parsePathwayMetaboliteDrugRelationships(*fhandler*)

stringParser.py

textminingParser.py

uniprotParser.py

databases_controller.py

Experimental Data

Experimental Data Parsers

clinicalParser.py

parser(*projectId*, *type='clinical'*)

project_parser(*projectId*, *config*, *directory*)

experimental_design_parser(*projectId*, *config*, *directory*)

clinical_parser(*projectId*, *config*, *clinical_directory*)

parse_dataset(*projectId*, *configuration*, *dataDir*, *key='project'*)
This function parses clinical data from subjects in the project Input: uri of the clinical data file. Format: Subjects as rows, clinical variables as columns Output: pandas DataFrame with the same input format but the clinical variables mapped to the right ontology (defined in config), i.e. type = -40 -> SNOMED CT

extract_project_info(*project_data*)

extract_responsible_rels(*project_data*, *separator='|'*)

extract_participant_rels(*project_data*, *separator='|'*)

extract_project_tissue_rels(*project_data*, *separator='|'*)

extract_project_disease_rels(*project_data*, *separator='|'*)

extract_project_intervention_rels(*project_data*, *separator='|'*)

extract_project_rels(*project_data*, *separator='|'*)

extract_timepoints(*project_data*, *separator='|'*)

extract_project_subject_rels(*projectId*, *design_data*)

extract_subject_identifiers(*design_data*)

extract_biosample_identifiers(*design_data*)

extract_analytical_sample_identifiers(*design_data*)

extract_biological_sample_subject_rels(*design_data*)

extract_biological_sample_analytical_sample_rels(*design_data*)

extract_biological_samples_info(*clinical_data*)

extract_analytical_samples_info(*data*)

extract_biosample_analytical_sample_relationship_attributes(*clinical_data*)

extract_biological_sample_timepoint_rels(*clinical_data*)

extract_biological_sample_tissue_rels(*clinical_data*)

extract_subject_disease_rels(*clinical_data*, *separator='|'*)

extract_subject_intervention_rels(*clinical_data*, *separator='|'*)

extract_biological_sample_clinical_variables_rels(*clinical_data*)

proteomicsParser.py

wesParser.py

parser(*projectId*)

parseWESDataset(*projectId*, *configuration*, *dataDir*)

loadWESDataset(*uri*, *configuration*)
This function gets the molecular data from a Whole Exome Sequencing experiment. Input: uri of the processed file resulting from the WES analysis pipeline. The resulting Annovar annotated VCF file from Mutect (sampleID_mutect_annovar.vcf) Output: pandas DataFrame with the columns and filters defined in config.py

extractWESRelationships(*data*, *configuration*)

experiments_controller.py

User Creation

users_controller.py

CKG Builder

create_user.py

importer.py

loader.py

builder.py

builder_utils.py

readDataset(*uri*)

readDataFromCSV(*uri*, *sep=','*, *header=0*, *comment=None*)
Read the data from csv file

readDataFromTXT(*uri*)
Read the data from tsv or txt file

readDataFromExcel(*uri*)
Read the data from Excel file

get_files_by_pattern(*regex_path*)

get_extra_pairs(*directory*, *extra_file*)

parse_contents(*contents*, *filename*)
Reads binary string files and returns a Pandas DataFrame.

export_contents(*data*, *dataDir*, *filename*)
Export Pandas DataFrame to file, with UTF-8 endocing.

parse_mztab_filehandler(*mztabf*)

parse_mztab_file(*mztab_file*)

parse_sdrf_filehandler(*sdrf_fh*)

convert_ckg_to_sdrf(*df*)

convert_sdrf_to_ckg(*df*)

convert_ckg_clinical_to_sdrf(*df*)

convert_sdrf_file_to_ckg(*file_path*)

write_relationships(*relationships*, *header*, *outputfile*)
Reads a set of relationships and saves them to a file.

Parameters

- **relationships** (*set*) – set of tuples with relationship data: source node, target node, relationship type, source and other attributes.
- **header** (*list*) – list of column names.
- **outputfile** (*str*) – path to file to be saved (including filename and extention).

write_entities(*entities*, *header*, *outputfile*)
Reads a set of entities and saves them to a file.

Parameters

- **entities** (*set*) – set of tuples with entities data: identifier, label, name and other attributes.
- **header** (*list*) – list of column names.
- **outputfile** (*str*) – path to file to be saved (including filename and extention).

get_config(*config_name*, *data_type='databases'*)
Reads YAML configuration file and converts it into a Python dictionary.

Parameters

- **config_name** (*str*) – name of the configuration YAML file.
- **data_type** (*str*) – configuration type ('databases' or 'ontologies').

Returns Dictionary.

Note: Use this function to obtain configuration for individual database/ontology parsers.

---

expand_cols(*data*, *col*, *sep=';'*)
Expands the rows of a dataframe by splitting the specified column

Parameters

- **data** – dataframe to be expanded
- **col** (*str*) – column that contains string to be expanded (i.e. 'P02788;E7EQB2;E7ER44;P02788-2;C9JCF5')
- **sep** (*str*) – separator (i.e. ';')

Returns expanded pandas dataframe

setup_config(*data_type='databases'*)
Reads YAML configuration file and converts it into a Python dictionary.

Parameters **data_type** – configuration type ('databases', 'ontologies', 'experiments' or 'builder').

Returns Dictionary.

Note: This function should be used to obtain the configuration for databases_controller.py, ontologies_controller.py, experiments_controller.py and builder.py.

---

list_ftp_directory(*ftp_url*, *user=''*, *password=''*)
Lists all files present in folder from FTP server.

Parameters

- **ftp_url** (*str*) – link to access ftp server.
- **user** (*str*) – username to access ftp server if required.
- **password** (*str*) – password to access ftp server if required.

Returns List of files contained in ftp server folder provided with ftp_url.

setup_logging(*path='log.config'*, *key=None*)
Setup logging configuration.

Parameters

- **path** (*str*) – path to file containing configuration for logging file.
- **key** (*str*) – name of the logger.

Returns Logger with the specified name from 'key'. If key is *None*, returns a logger which is the root logger of the hierarchy.

download_from_ftp(*ftp_url*, *user*, *password*, *to*, *file_name*)

download_PRIDE_data(*pxd_id*, *file_name*, *to='.'*, *user=''*, *password=''*, *date_field='publicationDate'*)
This function downloads a project file from the PRIDE repository

Parameters

- **pxd_id** (*str*) – PRIDE project identifier (id. PXD013599).
- **file_name** (*str*) – name of the file to dowload
- **to** (*str*) – local directory where the file should be downloaded
- **user** (*str*) – username to access biomedical database server if required.
- **password** (*str*) – password to access biomedical database server if required.
- **date_field** (*str*) – projects deposited in PRIDE are search based on date, either submissionData or publicationDate (default)

downloadDB(*databaseURL*, *directory=None*, *file_name=None*, *user=''*, *password=''*, *avoid_wget=False*)
This function downloads the raw files from a biomedical database server when a link is provided.

Parameters

- **databaseURL** (*str*) – link to access biomedical database server.
- **directory** (*str or None*) –
- **file_name** (*str or None*) – name of the file to dowload. If None, 'databaseURL' must contain filename after the last '/'.
- **user** (*str*) – username to access biomedical database server if required.

- **password** (*str*) – password to access biomedical database server if required.

searchPubmed(*searchFields*, *sortby='relevance'*, *num='10'*, *resultsFormat='json'*)
Searches PubMed database for MeSH terms and other additional fields ('searchFields'), sorts them by relevance and returns the top 'num'.

Parameters

- **searchFields** (*list*) – list of search fields to query for.
- **sortby** (*str*) – parameter to use for sorting.
- **num** (*str*) – number of PubMed identifiers to return.
- **resultsFormat** (*str*) – format of the PubMed result.

Returns Dictionary with total number of PubMed ids, and top 'num' ids.

is_number(*s*)
This function checks if given input is a float and returns True if so, and False if it is not.

Parameters **s** – input

Returns Boolean.

getMedlineAbstracts(*idList*)
This function accesses NCBI over the WWWW and returns Medline data as a handle object, which is parsed and converted to a Pandas DataFrame.

Parameters **idList** (*str or list*) – single identifier or comma-delimited list of identifiers. All the identifiers must be from the database PubMed.

Returns Pandas DataFrame with columns: 'title', 'authors', 'journal', 'keywords', 'abstract', 'PMID' and 'url'.

remove_directory(*directory*)

listDirectoryFiles(*directory*)
Lists all files in a specified directory.

Parameters **directory** (*str*) – path to folder.

Returns List of file names.

listDirectoryFolders(*directory*)
Lists all directories in a specified directory.

Parameters **directory** (*str*) – path to folder.

Returns List of folder names.

listDirectoryFoldersNotEmpty(*directory*)
Lists all directories in a specified directory.

Parameters **directory** (*str*) – path to folder.

Returns List of folder names.

checkDirectory(*directory*)
Checks if given directory exists and if not, creates it.

Parameters **directory** (*str*) – path to folder.

flatten(*t*)
Code from: https://gist.github.com/shaxbee/0ada767debf9eefbdb6e Acknowledgements: Zbigniew Mandziejewicz (shaxbee) Generator flattening the structure

```
>>> list(flatten([2, [2, (4, 5, [7], [2, [6, 2, 6, [6], 4]], 6)]]))
[2, 2, 4, 5, 7, 2, 6, 2, 6, 6, 4, 6]
```

pretty_print (*data*)
This function provides a capability to "pretty-print" arbitrary Python data structures in a forma that can be used as input to the interpreter. For more information visit https://docs.python.org/2/library/pprint.html.

Parameters **data** – python object.

convertOBOtoNet (*ontologyFile*)
Takes an .obo file and returns a NetworkX graph representation of the ontology, that holds multiple edges between two nodes.

Parameters **ontologyFile** (*str*) – path to ontology file.

Returns NetworkX graph.

getCurrentTime ()
Returns current date (Year-Month-Day) and time (Hour-Minute-Second).

Returns Two strings: date and time.

convert_bytes (*num*)
This function will convert bytes to MB.... GB... etc.

Parameters **num** – float, integer or pandas.Series.

copytree (*src*, *dst*, *symlinks=False*, *ignore=None*)

file_size (*file_path*)
This function returns the file size.

Parameters **file_path** (*str*) – path to file.

Returns Size in bytes of a plain file.

Return type str

buildStats (*count*, *otype*, *name*, *dataset*, *filename*, *updated_on=None*)
Returns a tuple with all the information needed to build a stats file.

Parameters

- **count** (*int*) – number of entities/relationships.
- **otype** (*str*) – 'entity' or 'relationsgips'.
- **name** (*str*) – entity/relationship label.
- **dataset** (*str*) – database/ontology.
- **filename** (*str*) – path to file where entities/relationships are stored.

Returns Tuple with date, time, database name, file where entities/relationships are stored, file size, number of entities/relationships imported, type and label.

unrar (*filepath*, *to*)
Decompress RAR file :param str filepath: path to rar file :param str to: where to extract all files

unzip_file (*filepath*, *to*)
Decompress zipped file :param str filepath: path to zip file :param str to: where to extract all files

compress_directory (*folder_to_backup*, *dest_folder*, *file_name*)
Compresses folder to .tar.gz to create data backup archive file.

Parameters

- **folder_to_backup** (*str*) – path to folder to compress and backup.
- **dest_folder** (*str*) – path where to save compressed folder.
- **file_name** (*str*) – name of the compressed file.

read_gzipped_file (*filepath*)
Opens an underlying process to access a gzip file through the creation of a new pipe to the child.

Parameters **filepath** (*str*) – path to gzip file.

Returns A bytes sequence that specifies the standard output.

parse_fasta (*file_handler*)
Using BioPython to read fasta file as SeqIO objects

Parameters **file_handler** (*file_handler*) – opened fasta file

Return iterator records iterator of sequence objects

batch_iterator (*iterator*, *batch_size*)
Returns lists of length batch_size.

This can be used on any iterator, for example to batch up SeqRecord objects from Bio.SeqIO.parse(...), or to batch Alignment objects from Bio.AlignIO.parse(...), or simply lines from a file handle.

This is a generator function, and it returns lists of the entries from the supplied iterator. Each list will have batch_size entries, although the final list may be shorter.

Parameters

- **iterator** (*iterator*) – batch to be extracted
- **batch_size** (*integer*) – size of the batch

Return list batch list with the batch elements of size batch_size

source: https://biopython.org/wiki/Split_large_file

mapping.py

9.1.3 Report Manager (report_manager)

Apps

basicApp.py

class BasicApp (*title*, *subtitle*, *description*, *page_type*, *layout=[]*, *logo=None*, *footer=None*)
Bases: object

Defines what an App is in the report_manager. Other Apps will inherit basic functionality from this class. Attributes: Title, subtitle, description, logo, footer.

property title

property subtitle

property description

property page_type

property logo

property footer

property layout

add_to_layout(*section*)

extend_layout(*sections*)

get_HTML_title()

get_HTML_subtitle()

get_HTML_description()

add_basic_layout()
Calls class functions to setup the layout: title, subtitle, description, logo and footer.

build_page()
Builds page basic layout.

dataUploadApp.py

class DataUploadApp(*title*, *subtitle*, *description*, *layout=[]*, *logo=None*, *footer=None*)
Bases: *ckg.report_manager.apps.basicApp.BasicApp*

Defines what the dataUpload App is in the report_manager. Used to upload experimental and clinical data to correct project folder.

> **Warning:** There is a size limit of 55MB. Files bigger than this will have to be moved manually.

buildPage()
Builds page with the basic layout from *basicApp.py* and adds relevant Dash components for project data upload.

dataUpload.py

homepageApp.py

homepageStats.py

importsApp.py

imports.py

initialApp.py

loginApp.py

class LoginApp(*title*, *subtitle*, *description*, *layout=[]*, *logo=None*, *footer=None*)
Bases: *ckg.report_manager.apps.basicApp.BasicApp*

Defines the login App Enables user to access the reports

buildPage()
Builds page with the basic layout from *basicApp.py* and adds the login form.

projectApp.py

projectCreationApp.py

class ProjectCreationApp(*title*, *subtitle*, *description*, *layout=[]*, *logo=None*, *footer=None*)
Bases: *ckg.report_manager.apps.basicApp.BasicApp*

Defines what the project creation App is in the report_manager. Includes multiple fill in components to gather project information and metadata.

buildPage()
Builds page with the basic layout from *basicApp.py* and adds relevant Dash components for project creation.

projectCreation.py

app.py

dataset.py

index.py

knowledge.py

project.py

report.py

user.py

utils.py

copy_file_to_destination(*cfile*, *destination*)

send_message_to_slack_webhook(*message*, *message_to*, *username='albsantosdel'*)

send_email(*message*, *subject*, *message_from*, *message_to*)

compress_directory(*name*, *directory*, *compression_format='zip'*)

get_markdown_date(*extra_text*)

convert_html_to_dash(*el*, *style=None*)

extract_style(*el*)

get_image(*figure*, *width*, *height*)

parse_html(*html_snippet*)

hex2rgb(*color*)

getNumberText(*num*)

get_rgb_colors(*n*)

get_hex_colors(*n*)

convert_html_to_pdf(*source_html*, *output_filename*)

worker.py

9.1.4 Analytics Core (analytics_core)

Analytics

analytics.py

wgcnaAnalysis.py

get_data(*data, drop_cols_exp=['subject', 'group', 'sample', 'index'], drop_cols_cli=['subject', 'group', 'biological_sample', 'index'], sd_cutoff=0*)
This function cleanes up and formats experimental and clinical data into similarly shaped dataframes.

Parameters

- **data** (*dict*) – dictionary with processed clinical and proteomics datasets.
- **drop_cols_exp** (*list*) – list of columns to drop from processed experimental (protemics/rna-seq/dna-seq) dataframe.
- **drop_cols_cli** (*list*) – list of columns to drop from processed clinical dataframe.

Returns Dictionary with experimental and clinical dataframes (keys are the same as in the input dictionary).

get_dendrogram(*df, labels, distfun='euclidean', linkagefun='ward', div_clusters=False, fcluster_method='distance', fcluster_cutoff=15*)
This function calculates the distance matrix and performs hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it.

Parameters

- **df** – pandas dataframe with samples/subjects as index and features as columns.
- **labels** (*list*) – labels for the leaves of the tree.
- **distfun** (*str*) – distance measure to be used ('euclidean', 'maximum', 'manhattan', 'canberra', 'binary', 'minkowski' or 'jaccard').
- **linkagefun** (*str*) – hierarchical/agglomeration method to be used ('single', 'complete', 'average', 'weighted', 'centroid', 'median' or 'ward').
- **div_clusters** (*bool*) – dividing dendrogram leaves into clusters (True or False).
- **fcluster_method** (*str*) – criterion to use in forming flat clusters.
- **fcluster_cutoff** (*int*) – maximum cophenetic distance between observations in each cluster.

Returns Dictionary of data structures computed to render the dendrogram. Keys: 'icoords', 'dcoords', 'ivl' and 'leaves'. If div_clusters is used, it will also return a dictionary of each cluster and respective leaves.

get_clusters_elements(*linkage_matrix, fcluster_method, fcluster_cutoff, labels*)
This function implements the generation of flat clusters from an hierarchical clustering with the same interface as scipy.cluster.hierarchy.fcluster.

Parameters

- **linkage_matrix** (*ndarray*) – hierarchical clustering encoded with a linkage matrix.

- **fcluster_method** (*str*) – criterion to use in forming flat clusters ('inconsistent', 'distance', 'maxclust', 'monocrit', 'maxclust_monocrit').
- **fcluster_cutoff** (*float*) – maximum cophenetic distance between observations in each cluster.
- **labels** (*list*) – labels for the leaves of the dendrogram.

Returns A dictionary where keys are the cluster numbers and values are the dendrogram leaves.

filter_df_by_cluster(*df*, *clusters*, *number*)
Select only the members of a defined cluster.

Parameters

- **df** – pandas dataframe with samples/subjects as index and features as columns.
- **clusters** (*dict*) – clusters dictionary from get_dendrogram function if div_clusters option was True.
- **number** (*int*) – cluster number (key).

Returns Pandas dataframe with all the features (columns) and samples/subjects belonging to the defined cluster (index).

df_sort_by_dendrogram(*df*, *Z_dendrogram*)
Reorders pandas dataframe by index and according to the dendrogram list of leaf nodes labels.

Parameters

- **df** – pandas dataframe with the labels to be reordered as index.
- **Z_dendrogram** (*dict*) – dictionary of data structures computed to render the dendrogram. Keys: 'icoords', 'dcoords', 'ivl' and 'leaves'.

Returns Reordered pandas dataframe.

get_percentiles_heatmap(*df*, *Z_dendrogram*, *bydendro=True*, *bycols=False*)
This function transforms the absolute values in each row or column (option 'bycols') into relative values.

Parameters

- **df** – pandas dataframe with samples/subjects as index and features as columns.
- **Z_dendrogram** (*dict*) – dictionary of data structures computed to render the dendrogram. Keys: 'icoords', 'dcoords', 'ivl' and 'leaves'.
- **bydendro** (*bool*) – if labels should be ordered according to dendrogram list of leaf nodes labels set to True, otherwise set to False.
- **bycols** (*bool*) – relative values calculated across rows (samples) then set to False. Calculation performed across columns (features) set to True.

Returns Pandas dataframe.

get_miss_values_df(*data*)
Proccesses pandas dataframe so missing values can be plotted in heatmap with specific color.

Parameters **data** – pandas dataframe.

Returns Pandas dataframe with missing values as integer 1, and originally valid values as NaN.

paste_matrices(*matrix1*, *matrix2*, *rows*, *cols*)
Takes two matrices with analog shapes and concatenates each value in matrix 1 with corresponding one in matrix 2, returning a single pandas dataframe.

Parameters

- **matrix1** (*ndarray*) – input 1
- **matrix2** (*ndarray*) – input 2

Returns Pandas dataframe.

cutreeDynamic(*distmatrix*, *linkagefun='average'*, *minModuleSize=50*, *method='hybrid'*, *deepSplit=2*, *pamRespectsDendro=False*, *distfun=None*)

This function implements the R cutreeDynamic wrapper in Python, provinding an access point for methods of adaptive branh pruning of hierarchical clustering dendrograms.

Parameters

- **data** – pandas dataframe.
- **distfun** (*str*) – distance measure to be used ('euclidean', 'maximum', 'manhattan', 'canberra', 'binary', 'minkowski' or 'jaccard').
- **linkagefun** (*str*) – hierarchical/agglomeration method to be used ('single', 'complete', 'average', 'weighted', 'centroid', 'median' or 'ward').
- **minModuleSize** (*int*) – minimum module size.
- **method** (*str*) – method to use ('hybrid' or 'tree').
- **deepSplit** (*int*) – provides a rough control over sensitivity to cluster splitting, the higher the value (with 'hybrid' method) or if True (with 'tree' method), the more and smaller modules.
- **pamRespectsDendro** (*bool*) – only used for method 'hybrid'. Objects and small modules will only be assigned to modules that belong to the same branch in the dendrogram structure.

Returns Numpy array of numerical labels giving assignment of objects to modules. Unassigned objects are labeled 0, the largest module has label 1, next largest 2 etc.

build_network(*data*, *softPower=6*, *networkType='unsigned'*, *linkagefun='average'*, *method='hybrid'*, *minModuleSize=50*, *deepSplit=2*, *pamRespectsDendro=False*, *merge_modules=True*, *MEDissThres=0.4*, *verbose=0*)

Weighted gene network construction and module detection. Calculates co-expression similarity and adjacency, topological overlap matrix (TOM) and clusters features in modules.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **softPower** (*int*) – soft-thresholding power.
- **networkType** (*str*) – network type ('unsigned', 'signed', 'signed hybrid', 'distance').
- **linkagefun** (*str*) – hierarchical/agglomeration method to be used ('single', 'complete', 'average', 'weighted', 'centroid', 'median' or 'ward').
- **method** (*str*) – method to use ('hybrid' or 'tree').
- **minModuleSize** (*int*) – minimum module size.
- **pamRespectsDendro** (*bool*) – only used for method 'hybrid'. Objects and small modules will only be assigned to modules that belong to the same branch in the dendrogram structure.
- **merge_modules** (*bool*) – if True, very similar modules are merged.
- **MEDissThres** (*float*) – maximum dissimilarity (i.e., 1-correlation) that qualifies modules for merging.

- **verbose** (*int*) – integer level of verbosity. Zero means silent, higher values make the output progressively more and more verbose.

Paran int deepSplit provides a rough control over sensitivity to cluster splitting, the higher the value (with 'hybrid' method) or if True (with 'tree' method), the more and smaller modules.

Returns Tuple with TOM dissimilarity pandas dataframe, numpy array with module colors per experimental feature.

pick_softThreshold(*data*, *RsquaredCut=0.8*, *networkType='unsigned'*, *verbose=0*)

Analysis of scale free topology for multiple soft thresholding powers. Aids the user in choosing a proper soft-thresholding power for network construction.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **RsquaredCut** (*float*) – desired minimum scale free topology fitting index R^2.
- **networkType** (*str*) – network type ('unsigned', 'signed', 'signed hybrid', 'distance').
- **verbose** (*int*) – integer level of verbosity. Zero means silent, higher values make the output progressively more and more verbose.

Returns Estimated appropriate soft-thresholding power: the lowest power for which the scale free topology fit R^2 exceeds RsquaredCut.

Return type int

identify_module_colors(*matrix*, *linkagefun='average'*, *method='hybrid'*, *minModuleSize=30*, *deepSplit=2*, *pamRespectsDendro=False*)

Identifies co-expression modules and converts the numeric labels into colors.

Parameters

- **matrix** – dissimilarity structure as produced by R.stats dist.
- **minModuleSize** (*int*) – minimum module size.
- **deepSplit** (*int*) – provides a rough control over sensitivity to cluster splitting, the higher the value (with 'hybrid' method) or if True (with 'tree' method), the more and smaller modules.
- **pamRespectsDendro** (*bool*) – only used for method 'hybrid'. Objects and small modules will only be assigned to modules that belong to the same branch in the dendrogram structure.

Returns Numpy array of strings with module color of each experimental feature.

calculate_module_eigengenes(*data*, *modColors*, *softPower=6*, *dissimilarity=True*)

Calculates modules eigengenes to quantify co-expression similarity of entire modules.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **modColors** (*ndarray*) – array (numeric, character or a factor) attributing module colors to each feature in the experimental dataframe.
- **softPower** (*int*) – soft-thresholding power.
- **dissimilarity** – calculates dissimilarity of module eigengenes.

Returns Pandas dataframe with calculated module eigengenes. If dissimilarity is set to True, returns a tuple with two pandas dataframes, the first with the module eigengenes and the second with the eigengenes dissimilarity.

merge_similar_modules (*data*, *modColors*, *MEDissThres=0.4*, *verbose=0*)

Merges modules in co-expression network that are too close as measured by the correlation of their eigengenes.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **modColors** (*ndarray*) – array (numeric, character or a factor) attributing module colors to each feature in the experimental dataframe.
- **verbose** (*int*) – integer level of verbosity. Zero means silent, higher values make the output progressively more and more verbose.

Para, float MEDissThres maximum dissimilarity (i.e., 1-correlation) that qualifies modules for merging.

Returns Tuple containing pandas dataframe with eigengenes of the new merged modules, and array with module colors of each expeirmental feature.

calculate_ModuleTrait_correlation (*df_exp*, *df_traits*, *MEs*)

Correlates eigengenes with external traits in order to identify the most significant module-trait associations.

Parameters

- **df_exp** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **df_traits** – pandas dataframe containing clinical data, with samples/subjects as rows and clinical traits as columns.
- **MEs** – pandas dataframe with module eigengenes.

Returns Tuple with two pandas datafames, first the correlation between all module eigengenes and all clinical traits, second a dataframe with concatenated correlation and p-value used for heatmap annotation.

calculate_ModuleMembership (*data*, *MEs*)

For each module, calculates the correlation of the module eigengene and the feature expression profile (quantitative measure of module membership (MM)).

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **MEs** – pandas dataframe with module eigengenes.

Returns Tuple with two pandas dataframes, one with module membership correlations and another with p-values.

calculate_FeatureTraitSignificance (*df_exp*, *df_traits*)

Quantifies associations of individual experimental features with the measured clinical traits, by defining Feature Significance (FS) as the absolute value of the correlation between the feature and the trait.

Parameters

- **df_exp** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.

- **df_traits** – pandas dataframe containing clinical data, with samples/subjects as rows and clinical traits as columns.

Returns Tuple with two pandas dataframes, one with feature significance correlations and another with p-values.

get_FeaturesPerModule(*data*, *modColors*, *mode='dictionary'*)

Groups all experimental features by the co-expression module they belong to.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **modColors** (*ndarray*) – array (numeric, character or a factor) attributing module colors to each feature in the experimental dataframe.
- **mode** (*str*) – type of the value returned by the function ('dictionary' or 'dataframe').

Returns Depending on selected mode, returns a dictionary or dataframe with module color per experimental feature.

get_ModuleFeatures(*data*, *modColors*, *modules=[]*)

Groups and returns a list of the experimental features clustered in specific co-expression modules.

Parameters

- **data** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **modColors** (*ndarray*) – array (numeric, character or a factor) attributing module colors to each feature in the experimental dataframe.
- **modules** (*list*) – list of module colors of interest.

Returns List of lists with experimental features in each selected module.

get_EigengenesTrait_correlation(*MEs*, *data*)

Eigengenes are used as representative profiles of the co-expression modules, and correlation between them is used to quantify module similarity. Clinical traits are added to the eigengenes to see how the traits fir into the eigengen network.

Parameters

- **MEs** – pandas dataframe with module eigengenes.
- **data** – pandas dataframe containing clinical data, with samples/subjects as rows and clinical traits as columns.

Returns Tuple with two pandas dataframes, one with features and traits recalculates module eigengenes dissimilarity, and another with all the overall correlations.

kaplan_meierAnalysis.py

get_data_ready_for_km(*dfs_dict*, *args*)

group_data_based_on_marker(*df*, *marker*, *index_col*, *how*, *value*)

run_km(*data*, *time_col*, *event_col*, *group_col*, *args={}*)

get_km_results(*df*, *group_col*, *time_col*, *event_col*)

get_hazard_ratio_results(*df*, *group_col*, *time_col*, *event_col*)

Viz

viz.py

wgcnaFigures.py

get_module_color_annotation(*map_list*, *col_annotation=False*, *row_annotation=False*, *bygene=False*, *module_colors=[]*, *dendrogram=[]*)

This function takes a list of values, converts them into colors, and creates a new plotly object to be used as an annotation. Options module_colors and dendrogram only apply when map_list is a list of experimental features used in module eigenegenes calculation.

Parameters

- **map_list** (*list*) – dendrogram leaf labels.
- **col_annotation** (*bool*) – if True, adds color annotations as a row.
- **row_annotation** (*bool*) – if True, adds color annotations as a column.
- **bygene** (*bool*) – determines wether annotation colors have to be reordered to match dendrogram leaf labels.
- **module_colors** (*list*) – dendrogram leaf module color.
- **dendrogram** (*dict*) – dendrogram represented as a plotly object figure.

Returns Plotly object figure.

Note: map_list and module_colors must have the same length.

---

get_heatmap(*df*, *colorscale=None*, *color_missing=True*)

This function plots a simple Plotly heatmap.

Parameters

- **df** – pandas dataframe containing experimental data, with samples/subjects as rows and features as columns.
- **colorscale** (*list[list]*) – heatmap colorscale (e.g. [[0,'#67a9cf'],[0.5,'#f7f7f7'],[1,'#ef8a62']]). If colorscale is not defined, will take [[0, 'rgb(255,255,255)'], [1, 'rgb(255,51,0)']] as default.
- **color_missing** (*bool*) – if set to True, plots missing values as grey in the heatmap.

Returns Plotly object figure.

plot_labeled_heatmap(*df, textmatrix, title, colorscale=[[0, 'rgb(0,255,0)'], [0.5, 'rgb(255,255,255)'], [1, 'rgb(255,0,0)']], width=1200, height=800, row_annotation=False, col_annotation=False*)

This function plots a simple Plotly heatmap with column and/or row annotations and heatmap annotations.

Parameters

- **df** – pandas dataframe containing data to be plotted in the heatmap.
- **textmatrix** – pandas dataframe with heatmap annotations as values.
- **title** (*str*) – the title of the figure.
- **colorscale** (*list[list]*) – heatmap colorscale (e.g. [[0,'rgb(0,255,0)'],[0.5,'rgb(255,255,255)'],[1,'rgb(255,0,0)']])

- **width** (*int*) – the width of the figure.
- **height** (*int*) – the height of the figure.
- **row_annotation** (*bool*) – if True, adds a color-coded column at the left of the heatmap.
- **col_annotation** (*bool*) – if True, adds a color-coded row at the bottom of the heatmap.

Returns Plotly object figure.

plot_dendrogram_guidelines(*Z_tree*, *dendrogram*)
This function takes a dendrogram tree dictionary and its plotly object and creates shapes to be plotted as vertical dashed lines in the dendrogram.

Parameters

- **Z_tree** (*dict*) – dictionary of data structures computed to render the dendrogram. Keys: 'icoords', 'dcoords', 'ivl' and 'leaves'.
- **dendrogram** – dendrogram represented as a plotly object figure.

Returns List of dictionaries.

plot_intramodular_correlation(*MM*, *FS*, *feature_module_df*, *title*, *width=1000*, *height=800*)
This function uses the Feature significance and Module Membership measures, and plots a multi-scatter plot of all modules against all clinical traits.

Parameters

- **MM** – pandas dataframe with module membership data
- **FS** – pandas dataframe with feature significance data
- **feature_module_df** – pandas DataFrame of experimental features and module colors (use mode='dataframe' in get_FeaturesPerModule)
- **title** (*str*) – plot title
- **width** (*int*) – plot width
- **height** (*int*) – plot height

Returns Plotly object figure.

Example:

```
plot = plot_intramodular_correlation(MM, FS, feature_module_df, title='Plot',␣
↪width=1000, height=800):
```

Note: There is a limit in the number of subplots one can make in Plotly. This function limits the number of modules shown to 5.

---

plot_complex_dendrogram(*dendro_df*, *subplot_df*, *title*, *dendro_labels=[]*, *distfun='euclidean'*, *linkagefun='average'*, *hang=0.04*, *subplot='module colors'*, *subplot_colorscale=[]*, *color_missingvals=True*, *row_annotation=False*, *col_annotation=False*, *width=1000*, *height=800*)
This function plots a dendrogram with a subplot below that can be a heatmap (annotated or not) or module colors.

Parameters

- **dendro_df** – pandas dataframe containing data used to generate dendrogram, columns will result in dendrogram leaves.

- **subplot_df** – pandas dataframe containing data used to generate plot below dendrogram.
- **title** (*str*) – the title of the figure.
- **dendro_labels** (*list*) – list of strings for dendrogram leaf nodes labels.
- **distfun** (*str*) – distance measure to be used ('euclidean', 'maximum', 'manhattan', 'canberra', 'binary', 'minkowski' or 'jaccard').
- **linkagefun** (*str*) – hierarchical/agglomeration method to be used ('single', 'complete', 'average', 'weighted', 'centroid', 'median' or 'ward').
- **hang** (*float*) – height at which the dendrogram leaves should be placed.
- **subplot** (*str*) – type of plot to be shown below the dendrogram (´module colors´ or ´heatmap´).
- **subplot_colorscale** (*list*) – colorscale to be used in the subplot.
- **color_missingvals** (*bool*) – if set to *True*, plots missing values as grey in the heatmap.
- **row_annotation** (*bool*) – if *True*, adds a color-coded column at the left of the heatmap.
- **col_annotation** (*bool*) – if *True*, adds a color-coded row at the bottom of the heatmap.
- **width** (*int*) – the width of the figure.
- **height** (*int*) – the height of the figure.

Returns Plotly object figure.

Dendrogram.py

plot_dendrogram(*Z_dendrogram*, *cutoff_line=True*, *value=15*, *orientation='bottom'*, *hang=30*, *hide_labels=False*, *labels=None*, *colorscale=None*, *hovertext=None*, *color_threshold=None*)

Modified version of Plotly _dendrogram.py that returns a dendrogram Plotly figure object with cutoff line.

Parameters

- **Z_dendrogram** (*ndarray*) – Matrix of observations as array of arrays
- **cutoff_line** (*boolean*) – plot distance cutoff line
- **value** (*float or int*) – dendrogram distance for cutoff line
- **orientation** (*str*) – 'top', 'right', 'bottom', or 'left'
- **hang** (*float*) – dendrogram distance of leaf lines
- **hide_labels** (*boolean*) – show leaf labels
- **labels** (*list*) – List of axis category labels(observation labels)
- **colorscale** (*list*) – Optional colorscale for dendrogram tree
- **hovertext** (*list[list]*) – List of hovertext for constituent traces of dendrogram clusters
- **color_threshold** (*double*) – Value at which the separation of clusters will be made

Returns Plotly figure object

Example:

```
figure = plot_dendrogram(dendro_tree, hang=0.9, cutoff_line=False)
```

class Dendrogram(*Z_dendrogram*, *orientation='bottom'*, *hang=1*, *hide_labels=False*, *labels=None*, *colorscale=None*, *hovertext=None*, *color_threshold=None*, *width=inf*, *height=inf*, *xaxis='xaxis'*, *yaxis='yaxis'*)

Bases: object

Refer to plot_dendrogram() for docstring.

get_color_dict(*colorscale*)

Returns colorscale used for dendrogram tree clusters.

Parameters **colorscale** (*list*) – colors to use for the plot in rgb format

Return (dict) default colors mapped to the user colorscale

set_axis_layout(*axis_key*, *hide_labels*)

Sets and returns default axis object for dendrogram figure.

Parameters **axis_key** (*str*) – E.g., 'xaxis', 'xaxis1', 'yaxis', yaxis1', etc.

Return (dict) An axis_key dictionary with set parameters.

set_figure_layout(*width*, *height*, *hide_labels*)

Sets and returns default layout object for dendrogram figure.

Parameters

- **width** (*int*) – plot width
- **height** (*int*) – plot height
- **hide_labels** (*boolean*) – show leaf labels

Returns Plotly layout

get_dendrogram_traces(*Z_dendrogram*, *hang*, *colorscale*, *hovertext*, *color_threshold*)

Calculates all the elements needed for plotting a dendrogram.

Parameters

- **Z_dendrogram** (*ndarray*) – Matrix of observations as array of arrays
- **hang** (*float*) – dendrogram distance of leaf lines
- **colorscale** (*list*) – Color scale for dendrogram tree clusters
- **hovertext** (*list*) – List of hovertext for constituent traces of dendrogram

Return (tuple) Contains all the traces in the following order:

a. trace_list: List of Plotly trace objects for dendrogram tree

b. icoord: All X points of the dendrogram tree as array of arrays with length 4

c. dcoord: All Y points of the dendrogram tree as array of arrays with length 4

d. ordered_labels: leaf labels in the order they are going to appear on the plot

e. Z_dendrogram['leaves']: left-to-right traversal of the leaves

color_list.py

Code for handling color names and RGB codes.

This module is part of Swampy, and used in Think Python and Think Complexity, by Allen Downey.

http://greenteapress.com

make_color_dict(*colors='\n141 211 199\t\tturquoise\n31 120 180\t\tblue\n139 69 19\t\tsaddlebrown\n177 89 40\t\tbrown\n51 160 44\t\tgreen\n255 237 111\t\tyellow\n173 255 47\t\tgreenyellow\n255 0 0\t\tred\n255 255 255\t\twhite\n0 0 0\t\tblack\n255 192 203\t\tpink\n255 0 255\t\tmagenta\n160 32 240\t\tpurple\n210 180 140\t\ttan\n250 128 114\t\tsalmon\n166 206 227\t\tcyan\n25 25 112\t\tmidnightblue\n224 255 255\t\tlightcyan\n153 153 153 \t\tgrey60\n144 238 144\t\tlightgreen\n255 255 224\t\tlightyellow\n65 105 225\t\troyalblue\n139 0 0\t\tdarkred\n0 100 0\t\tdarkgreen\n0 206 209\t\tdarkturquoise\n169 169 169\t\tdarkgrey\n255 165 0\t\torange\n255 140 0\t\tdarkorange\n135 206 235\t\tskyblue\n70 130 180\t\tsteelblue\n175 238 238\t\tpaleturquoise\n238 130 238\t\tviolet\n85 107 47\t\tdarkolivegreen\n139 0 139\t\tdarkmagenta\n190 190 190\t\tgray\n190 190 190\t\tgrey\n'*)

Returns a dictionary that maps color names to RGB strings.

The format of RGB strings is '#RRGGBB'.

read_colors()

Returns color information in two data structures.

The format of RGB strings is '#RRGGBB'.

color_dict: map from color name to RGB string rgbs: list of (rgb, names) pairs, where rgb is an RGB code and names is a sorted list of color names

invert_dict(*d*)

Returns a dictionary that maps from values to lists of keys.

d: dict

returns: dict

R_wrapper.py

call_Rpackage(*call='function'*, *designation='aov'*)

R_matrix2Py_matrix(*r_matrix*, *index*, *columns*)

analytics_factory.py

utils.py

check_columns(*df*, *cols*)

mpl_to_html_image(*plot*, *width=800*)

generate_html(*network*)

This method gets the data structures supporting the nodes, edges, and options and updates the pyvis html template holding the visualization.

append_to_list (*mylist*, *myappend*)

neo4j_path_to_networkx (*paths*, *key='path'*)

neo4j_schema_to_networkx (*schema*)

networkx_to_cytoscape (*graph*)

networkx_to_gml (*graph*, *path*)

networkx_to_neo4j_document (*graph*)

json_network_to_gml (*graph_json*, *path*)

networkx_to_graphml (*graph*, *path*)

json_network_to_graphml (*graph_json*, *path*)

json_network_to_networkx (*graph_json*)

get_clustergrammer_link (*net*, *filename=None*)

generator_to_dict (*genvar*)

parse_html (*html_snippet*)

convert_html_to_dash (*el*, *style=None*)

hex2rgb (*color*)

get_rgb_colors (*n*)

get_hex_colors (*n*)

getMedlineAbstracts (*idList*)

9.1.5 Notebooks - development

vis.py

9.1.6 ckg_utils.py

read_ckg_config (*key=None*)

save_dict_to_yaml (*data*, *yaml_file*)

read_yaml (*yaml_file*)

get_queries (*queries_file*)

get_configuration (*configuration_file*)

get_configuration_variable (*configuration_file*, *variable*)

setup_logging (*path='log.config'*, *key=None*)
Setup logging configuration

listDirectoryFiles (*directory*)

listDirectoryFolders (*directory*)

checkDirectory (*directory*)

is_jsonable (*x*)

convert_dash_to_json (*dash_object*)

class NumpyEncoder(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class DictDFEncoder(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

CHAPTER

TEN

ABOUT CKG

10.1 Credits

10.1.1 Development Leads

- Alberto Santos Delgado (@albsantosdel)
- Ana Rita Colaço (@arcolaco)
- Annelaura Bach Nielsen (@aannelaura)

10.1.2 Core Committers

10.1.3 Contributors

10.2 Backers

We would like to thank the following people for supporting us in our efforts to maintain and improve the Clinical Knowledge Graph:

-
-

10.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

- Types of Contributions *Types of Contributions*
- Contributor Setup *Setting Up the Code for Local Development*
- Contributor Guidelines *Contributor Guidelines*
- Core Committer Guide *Core Committer Guide*

10.3.1 Types of Contributions

You can contribute in many ways:

Create Analysis or Visualization methods

If you develop new ways of analysing or visualizing data, please feel free to add to the Analytics Core.

Report Bugs

Report bugs at https://github.com/MannLabs/CKG/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- If you can, provide detailed steps to reproduce the bug.
- If you don't have steps to reproduce the bug, just note your observations in as much detail as you can. Questions to start a discussion about the issue are welcome.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "please-help" is open to whoever wants to implement it.

Please do not combine multiple feature enhancements into a single pull request.

Write Documentation

The Clinical Knowledge Graph could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

If you want to review your changes on the documentation locally, you can do:

```
$ cd docs/
$ make servedocs
```

This will compile the documentation, open it in your browser and start watching the files for changes, recompiling as you save.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/MannLabs/CKG/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

10.3.2 Setting Up the Code for Local Development

Here's how to set up `CKG` for local development.

1. Fork the `CKG` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:MannLabs/CKG.git
```

3. Install your local copy according to the "Getting Started" tutorials.
4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

10.3.3 Contributor Guidelines

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and describe it.
2. The pull request should work for Python 3.5, 3.6 and 3.7.

Coding Standards

- PEP8
- Functions over classes except in tests
- Quotes via http://stackoverflow.com/a/56190/5549
  - Use double quotes around strings that are used for interpolation or that are natural language messages
  - Use single quotes for small symbol-like strings (but break the rules if the strings contain quotes)
  - Use triple double quotes for docstrings and raw string literals for regular expressions even if they aren't needed.
  - Example:

```
LIGHT_MESSAGES = {
    'English': "There are %(number_of_lights)s lights.",
    'Pirate':  "Arr! Thar be %(number_of_lights)s lights."
}
def lights_message(language, number_of_lights):
    """Return a language-appropriate string reporting the light count."""
    return LIGHT_MESSAGES[language] % locals()
def is_pirate(message):
    """Return True if the given message sounds piratical."""
    return re.search(r"(?i)(arr|avast|yohoho)!", message) is not None
```

- Write new code in Python 3.

10.3.4 Core Committer Guide

Vision and Scope

Core committers, use this section to:

- Guide your instinct and decisions as a core committer
- Limit the codebase from growing infinitely

Command-Line and API Accessible

- Provides command-line utilities that launch a dash app to browse projects, statistics and others, create new users, and import and load data into the database.
- Extremely easy to use without having to think too hard
- Flexible for more complex use via optional arguments

Extensible

Being extendable by people with different ideas.

- Entirely function-based
- Aim for statelessness
- Lets anyone write more opinionated tools

Freedom for CKG users to build and extend.

- Community-based project, all contributions to improve and/or extend the code are welcome.

Inclusive

- Cross-platform support.
- Fixing Windows bugs even if it's a pain, to allow for use by the entire community.

Process: Pull Requests

If a pull request is untriaged:

- Look at the roadmap
- Set it for the milestone where it makes the most sense
- Add it to the roadmap

How to prioritize pull requests, from most to least important:

- Fixes for broken code. Broken means broken on any supported platform or Python version.
- Features.
- Bug fixes.
- Major edits to docs.
- Extra tests to cover corner cases.
- Minor edits to docs.

Ensure that each pull request meets all requirements in checklist.

Process: Issues

If an issue is a bug that needs an urgent fix, mark it for the next patch release. Then either fix it or mark as please-help.

For other issues: encourage friendly discussion, moderate debate, offer your thoughts.

New features require a +1 from 2 other core committers (besides yourself).

Process: Pull Request merging and HISTORY.md maintenance

If you merge a pull request, you're responsible for updating `AUTHORS.rst` and `HISTORY.rst`

When you're processing the first change after a release, create boilerplate following the existing pattern:

```
## x.y.z (Development)

The goals of this release are TODO: release summary of features

Features:

* Feature description, thanks to [@contributor](https://github.com/contributor) (#PR).

Bug Fixes:

* Bug fix description, thanks to [@contributor](https://github.com/contributor) (#PR).

Other changes:

* Description of the change, thanks to [@contributor](https://github.com/contributor)␣
↪(#PR).
```

Process: Accepting New Features Pull Requests

- Run the feature to generate the output.
- Attempt to include it in the standard pipeline and run an example project dataset.
- Merge the feature in.
- Update the history file.

note: Adding features doesn't give authors credit.

Process: Your own code changes

All code changes, regardless of who does them, need to be reviewed and merged by someone else. This rule applies to all the core committers.

Exceptions:

- Minor corrections and fixes to pull requests submitted by others.
- While making a formal release, the release manager can make necessary, appropriate changes.
- Small documentation changes that reinforce existing subject matter. Most commonly being, but not limited to spelling and grammar corrections.

Responsibilities

- Ensure cross-platform compatibility for every change that's accepted. Windows, Mac, Debian & Ubuntu Linux.
- Ensure that code that goes into core meets all requirements in this checklist: https://gist.github.com/audreyr/4feef90445b9680475f2
- Create issues for any major changes and enhancements that you wish to make. Discuss things transparently and get community feedback.
- Keep feature versions as small as possible, preferably one new feature per version.
- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds. Look at *Code of Conduct :ref:code-of-conduct*.

10.4 History

10.4.1 1.0b0 (2020-05-11)

- First release on GitHub.

Beta version of CKG for trial under real conditions, by community users.

10.5 Code of Conduct

Everyone interacting in the Clinical Knowledge Graph codebases, issue trackers, chat rooms, and mailing lists is expected to follow the PyPA Code of Conduct.

CHAPTER

ELEVEN

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

H

I

J

L

M

N

P

R

S

T

U

W