Clinical Knowledge Graph Documentation

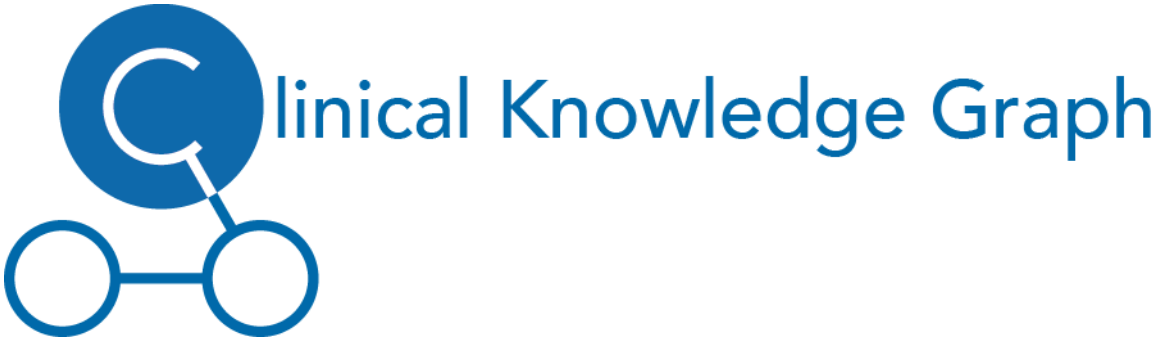
Release 1.0b1 BETA

Alberto Santos, Ana Rita Colaço, Annelaura B. Nielsen

Aug 03, 2020

CONTENTS

This web page contains the documentation for the Python code using **Sphinx**.



CHAPTER

ONE

CLINICAL KNOWLEDGE GRAPH

version: 1.0b1 BETA

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

1.1 Abstract

./src/report_manager/assets/abstract.png

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

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 ReadTheDocs 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.
- 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

Table 1 – continued from previous page

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
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

CHAPTER

TWO

FIRST STEPS

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

- **Getting started**: *With Requirements* | *With Neo4j* | *With Clinical Knowledge Graph* | *With Windows* | *With Docker*

2.1 Getting Started with some Requirements

The following instructions on installation of software requirements and setting up the Clinical Knowledge graph, are optimised for operating systems MacOS and Linux. For more detailed instructions on how to set up the CKG in Windows, please go to *Getting started with Windows*.

2.1.1 Java

Before starting setting up Neo4j and, later on, the Clinical Knowledge Graph, it is very important that you have *Java* installed in your machine, including **Java SE Runtime Environment**.

Different versions of a Neo4j database can have different requirements. For example, Neo4j 3.5 versions require Oracle Java 8, while Neo4j 4.0 versions already require Oracle Java 11. When using a new version of Neo4j, always remember to read the respective Operations Manual, and check for the software requirements.

To check if you already have **Java SE Development Kit** installed, run `java -version` in your terminal window. This should print out three lines similar to the following, with possible variation in the version:

```
java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)
```

Running `/usr/libexec/java_home` in the terminal should print out a path like `/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home`. Otherwise, please follow the steps below:

1. Go to `https://www.oracle.com/java/technologies/javase-downloads.html` and download the version that fits your Neo4j version and OS requirements.
2. Install the package.
3. Run `/usr/libexec/java_home` in the terminal to make sure the *Java* package has been installed in `/Library/Java/JavaVirtualMachines/`.

2.1.2 R

Another essential package for the functioning of the Clinical Knowledge Graph is R.

Make sure you have installed **R version >= 3.5.2**:

```
$ 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('flashClust','WGCNA', 'samr'), dependencies=TRUE, repos='http://
↪cran.rstudio.com/')
```

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

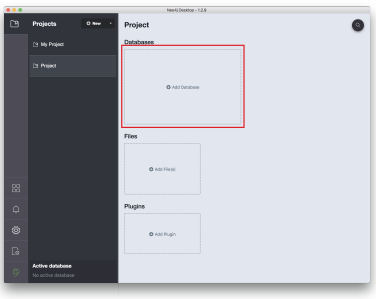
Warning: In Mac OS, make sure you have **XQuartz** installed, as well as **Xcode**. For more information on how to install R on OS X, you can follow this link.

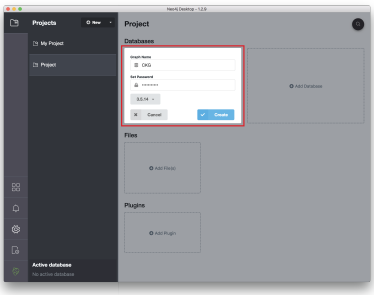
Now that you are all set, you can move on and start with Neo4j.

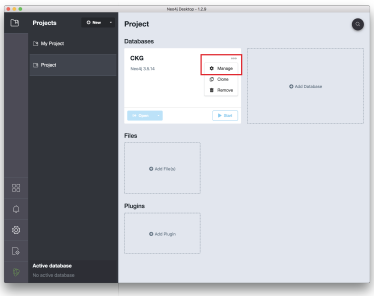
2.2 Getting Started with Neo4j

Getting started with Neo4j is easy.

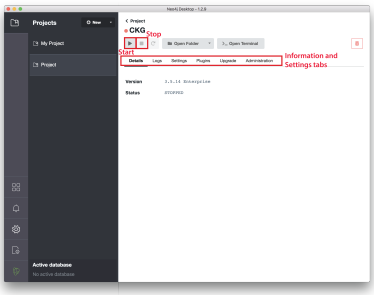
First download a copy of the Neo4j desktop version from the Neo4j download page. The Community Edition of the software is free but a sign up is required. Once the file has downloaded, you can install Neo4j by following the instructions automatically opened in the browser.

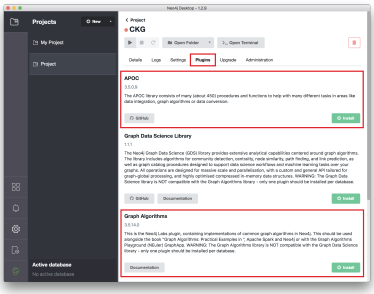


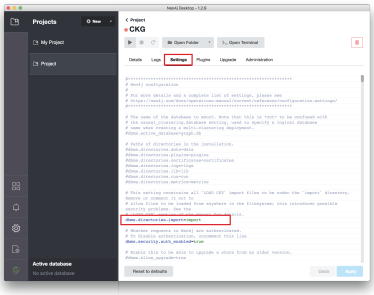




Open the Neo4j Desktop App and create a database by clicking *Add graph*, followed by *Create a Local Graph*, using the password "NeO4J". Now that your database is created:







1. Click *Manage* and then *Plugins*. Install "**APOC**" and "**GRAPH ALGORITHMS**".
2. Click the tab *Settings*, and comment the option `dbms.directories.import=import` by adding `#` at the beginning of the line.
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!

Note: Be aware, at the time of release of this version, "**GRAPH ALGORITHMS**" was only available for Neo4j database version **=< 3.14**.

2.3 Getting Started with the CKG Build

Setting up the Clinical Knowledge Graph is straightforward. Assuming you have **Python 3.6** already installed and added to `PATH`, you can choose to create a virtual environment where all the packages with the specific versions will be installed. To do so, use Virtualenv.

To check which Python version is currently installed:

```
$ python3.6 --version
```

And where this Python version is:

```
$ which python3.6
```

If this does not correspond to the correct Python version you want to run, you can create a shell alias in the bash file:

1. Open the bash file:

```
$ vi ~/.bash_profile
```

1. Add at the end of the file:

```
alias python3.6="/path/to/correct/python3.6"
```

1. Save and close the bash file
2. Make the alias available in the current session:

```
$ source ~/.bash_profile
```

Note: If you don't have **Python 3.6** installed, download the Python 3.6 version appropriate for your machine, and run the installer package. Python should be installed in `/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6` and also found in `/usr/local/bin/python3.6`.

2.3.1 Create a virtual environment

Virtualenv is not installed by default on Macbook machines. To install it, run:

```
$ python3 -m pip install virtualenv
```

To create a new virtual environment using a custom version of Python, follow the steps:

1. Take note of the full path to the Python version you would like to use inside the virtual environment.
2. Navigate to the directory where you would like your virtual environment to be (e.g. user's root).
3. Create the virtual environment at the same time you specify the version of Python you wish to use. `env_name` is the name of the virtual environment and can be set to anything you like.

```
$ virtualenv -p /path/to/python env_name
```

1. Activate the virtual environment by running:

```
$ source path/to/env_name/bin/activate
```

After this, the name of the virtual environment will now appear on the left of the prompt:

```
(env_name) username$
```

If you are finished working in the virtual environment for the moment, you can deactivate it by running:

```
$ deactivate
```

2.3.2 Setting up the Clinical Knowledge Graph

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

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

Once this is finished, you can find all the Python modules necessary to run the Clinical Knowledge Graph in `requirements.txt`. To install all the packages required, simply run:

```
$ cd CKG/
$ pip3 install --upgrade pip
$ pip3 install --ignore-installed -r requirements.txt
```

Warning: Make sure the virtual environment previously created is active before installing `requirements.txt`.

Now that all the packages are correctly installed, you will have to create the appropriate directory architecture within the local copy of the cloned repository:

```
$ python setup_CKG.py
$ python setup_config_files.py
```

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.

2.3.3 Add CKG to *.bashrc*

In order run the the Clinical Knowledge Graph, add the path to the code to your `.bashrc` (or `.bash_profile`):

1. Open the .bashrc file.
2. Add the following lines to the file and save it:

```
PYTHONPATH="${PYTHONPATH}:/path/to/folder/CKG/src/"
export PYTHONPATH
```

Notice that the path should always finish with "/CKG/src/".

1. To reload the bash file, first deactivate the virtual environment, reload ~/.bashrc, and activate the virtual environment again:

```
$ deactivate
$ source ~/.bashrc
$ source path/to/env_name/bin/activate
```

2.3.4 Build Neo4j graph database

The building of the CKG database is thoroughly automated. 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.

- 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*).

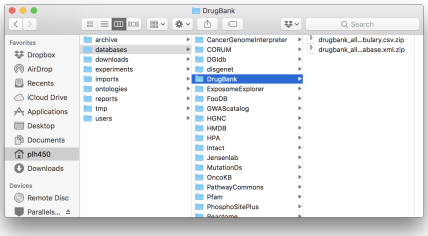


Fig. 2: DrugBank database folder.

- SNOMED-CT: *Download RF2 Files Now!*.

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.

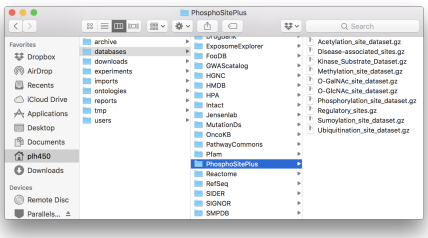


Fig. 3: PhosphoSitePlus database folder.

Warning: These three databases require login and authentication. To sign up go to PSP Sign up, DrugBank Sign up and SNOMED-CT Sign up. In the 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.

The last step is to build the database, which can be done using the `builder.py` module or a `dump file`.

From the provided 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 both files (`ckg_080520.dump` and `data.tar.gz`).

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

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

```
$ cd /path/to/neo4jDatabases/database-identifier/installation-x.x.x/
$ mkdir backups
$ mkdir backups/graph.db
$ cp 2019-11-04.dump backups/graph.db/.
```

2. 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_080520.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.

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

Be aware the database contained in the dump file **does NOT** include the licensed databases (**PhosphoSitePlus**, **DrugBank** and **SNOMED-CT**).

To add the missing ontology and databases, as well as their dependencies (relationships to other nodes), please manually download the files as explained in *Build Neo4j graph database*, unzip the downloaded file `data.tar.gz` and place its contents in `CKG/data/`. The folder `data` should look like the figure depicted.

Once this is done, run the following commands:

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

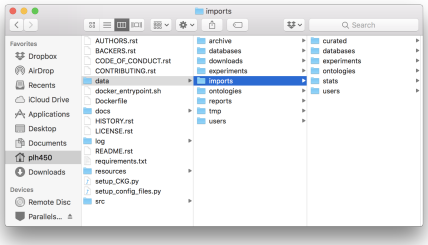


Fig. 4: Final CKG/data folder architecture.

Note: Remember of replace the `username` in each command, with your own neo4j username.

From builder.py

To build the graph database, run `builder.py`:

```
$ cd src/graphdb_builder/builder
$ python builder.py -b full -u neo4j
```

Warning: Before running `builder.py`, please make sure your Neo4j graph is running. The builder will fail otherwise.

This action will take approximately 6 hours but depending on a multitude of factors, it can take up to 10 hours.

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.

2.4 Getting started with Windows

In this section we describe how to install all the necessary requirements and set up the Clinical Knowledge Graph on a Windows operating system.

2.4.1 Java

Similarly to MacOS and Linux, Windows will also need a **Java** installation (Java SE Runtime Environment and Java SE Development Kit).

Be aware that different versions of a Neo4j database can have different requirements. For example, Neo4j 3.5 versions require Oracle Java 8, while Neo4j 4.0 versions already require Oracle Java 11. When using a new version of Neo4j, always remember to read the respective Operations Manual, and check for the software requirements.

By default Java should be installed on the Windows 10. If this is not your case, please follow this tutorial to install it.

2.4.2 Neo4j

The installation of Neo4j on Windows follows the same steps as *Getting Started with Neo4j*:

1. Download a copy of the Neo4j desktop version from the Neo4j download page.
2. Install Neo4j by following the instructions automatically opened in the browser.
3. Open the Neo4j Desktop App and create a database by clicking *Add graph*, followed by *Create a Local Graph*, using the password "NeO4J".
4. Click *Manage* and then *Plugins*. Install "**APOC**" and "**GRAPH ALGORITHMS**".
5. Click the tab *Settings*, and comment the option `dbms.directories.import=import` by adding `#` at the beginning of the line.
6. Click *Apply* at the bottom of the window.
7. Start the Graph by clicking the play sign, at the top of the window.

To check for errors, please go to tab *Logs*.

2.4.3 Python

Installing Python 3.6.8 is an essential part of setting up the CKG. To do so, simply use the link to download a python 3.6 version executable installer (e.g. Python 3.6.8 - Dec. 24, 2018). Follow the steps and select the option `Add Python 3.6 to PATH`.

Once finished, you can check if python was successfully added to PATH, by opening the environment variables window:

1. Go to the Windows menu, right-click on *Computer* and click on *Properties*.
2. From the computer properties dialog, select `Advanced system settings` on the left panel. And from there, click on *Environment variables* button.
3. In the top half of the new window (**User variables**), look for a variable with the name `Path`. If you select it and click *Edit*, the full path to Python36 should be included in the variable value. Besides this, you can also add the paths to python 3.6 `site-packages` and `Scripts` folders, to the `Path` variable.
4. When you are done, click *OK* to save, and click *OK* in all the opened windows.

2.4.4 Microsoft Visual C++ Build Tools

Running python on Windows can sometimes result in the following error:

```
error Microsoft Visual C++ 14.0 is required
```

To fix this error, you will need to download and install Microsoft Build Tools for Visual Studio.

Once installed, click `Workloads`, select all the packages available and install them. This will require several Gigabytes of disk space so, as an alternative and if your machine has limited space, you can install only `C++ build tools` under `Workloads`, and `Windows 10 SDK` and the latest version of `MSVC v142 - VS 2019 C++ x64/x86 build tools` under `Individual Components`.

The build tools allow Python packages to be built in Windows, from the command line (MSVC cl.exe module is used as a C/C++ compiler).

2.4.5 R

Another essential package for the functioning of the Clinical Knowledge Graph is R.

You can check if an **R version >= 3.5.2** is already installed by running:

```
> where R
```

If R is not installed in your machine, please follow these tutorial.

In order to simplify calling R from the command prompt, you can choose to add it to `PATH` and to the environment variables. To do so, follow the steps bellow:

1. Go to the Windows menu, right-click on *Computer* and click on *Properties*.
2. From the computer properties dialog, select `Advanced system settings` on the left panel. And from there, click on *Environment variables* button.
3. In the Environment variables dialog, click the *New* button in the top half of the dialog, to make a new user variable.
4. Give the variable name as `R` and the value is the path to the R executable, which is usually `C:\Program Files\R\R-4.0.0\bin\R.exe`.
5. In the bottom half of the Environment variables dialog, find the variable `Path`, select it and click *Edit*.
6. In the edit dialog window, add `;` to the end of the variable value followed by the R path used when creating the previous environmental variable.
7. Click *OK* to save, click *OK* and *OK* again to save the new variable and edit to `Path`.

To confirm that the environment variable is correctly set in command line type:

```
> echo %R%
```

This will print the path you used as value (e.g. `C:\Program Files\R\R-4.0.0\bin\R.exe`).

To run R from the command prompt, run:

```
> R
```

All R packages can be installed by simply initiating R (command prompt or R shell) and running:

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

Warning: If the install does not work (cannot write to library), run a new command prompt as administrator:

1. Go to the Windows menu, right-click on *Command Prompt* and select `Run as administrator`.

In this new prompt, launch R and run the previous R install packages.

2.4.6 Getting Started with the CKG Build

Setting up the Clinical Knowledge Graph is thoroughly described here. Assuming you have **Python 3.6** already installed, you can choose to create a virtual environment where all the packages with the specific versions will installed.

To check which Python version is currently installed, run in the command prompt (cmd.exe):

```
> python --version
```

And to find your this Python version is installed:

```
> where python
```

Create a virtual environment

To create a new Python virtual environment, you can choose to use `virtualenv` or `venv`.

The usage of `virtualenv` is exemplified in ref:*Create virtual environment*.

To use `venv`, open a command prompt (cmd.exe) window and type:

```
> python -m venv path\to\env_name
```

Note: `path\to\env_name` should be replaced with the relative or full path to where you want to place your virtual environment, while the `env_name` part is to be replaced with the name you want to attribute to the virtual environment.

Whichever way you create the virtual environment, the activation method is the same:

```
> path\to\env_name\Scripts\activate.bat
```

After this, the name of the virtual environment will now appear on the left of the prompt:

```
(env_name) C:\>
```

If you are finished working in the virtual environment for the moment, you can deactivate it by running:

```
> deactivate
```

Warning: Remember, every time you are working with the CKG, the virtual environment needs to be activated first.

Setting up the CKG

Once you have cloned the master branch of the CKG GitHub repository, all the Python packages necessary to run the Clinical Knowledge Graph can be found in `requirements.txt`.

Unfortunately, due to incompatibilities of the current versions `celery` and `rpy2` packages need to be removed from `requirements.txt` before installing all other packages.

To do so, open the mentioned file in your preferred text editor tool (e.g. Notepad) and add `#` in the beginning of the lines `celery==4.3.0` and `rpy2==3.0.5`. Save and close the file, making sure it is saved as a plain text file.

Warning: Part of the CKG functionality includes interfacing Python and R, and seemingly use R functions for data analysis. The python package `rpy2` is used as this interface and unfortunately, the current release of this package for Windows is not compatible with CKG. Installation of the CKG on Windows machines, will therefore **not** allow the usage of R packages (SAMR and WGCNA) within the CKG.

To install all the required packages, simply run:

```
> cd CKG\
> pip3 install --upgrade pip
> pip3 install --ignore-installed -r requirements.txt

.. warning:: Make sure the virtual environment previously created is active before␣
↪installing ``requirements.txt``.
```

When these packages are installed, you can proceed to install a functinal version of celery 4 for Windows:

```
> cd ..\
> git clone https://github.com/bstiel/celery-4-windows.git
> cd celery-4-windows
> pip install requirements.txt
```

Now that all the packages are correctly installed, you will have to create the appropriate directory architecture within the local copy of the cloned repository:

```
> python setup_CKG.py
> python setup_config_files.py
```

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.

Add CKG to environmental variables

In order to run CKG modules, the package needs to be added to the environmental variables.

1. Go to the Windows menu, right-click on *Computer* and click on *Properties*.
2. From the computer properties dialog, select `Advanced system settings` on the left panel. And from there, click on *Environment variables*.
3. In the Environment variables dialog, click *New* in the top half of the dialog, to make a new user variable
4. Give the variable name as `PYTHONPATH` and the value is the path to the CKG code directory, for example `C:\CKG\src`. Notice that the path should always finish with `\CKG\src`.
5. Click *OK* and *OK* again to save this variable.

To confirm that the environment variable is correctly set in command line type:

```
> echo %PYTHONPATH%
```

This will print the path you used as value (e.g. `C:\CKG\src`).

Build Neo4j graph database (Windows)

Building the CKG database in Windows follows the same steps as in MacOS and Linux so, from here on, please follow the tutorial *Build Neo4j graph database*.

2.5 Getting started with Docker (Testing)

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

To run the Docker, simply:

1. Allocate resources:

The docker build requires more resources (memory and disk) than the ones set as default. Make sure to allocate at least 8Gb memory and at least 60Gb of Disk space. To change these settings: Docker Preferences -> Resources.

2. Build the docker

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

3. Make sure to download manually the licensed databases (*Build Neo4j graph database*)

4. Run the docker

```
$ docker run -d --name ckgapp -d -v log:/CKG/log -v data:/CKG/data -e EXEC_MODE=
↪"minimal" --restart=always -p 8050:8050 -p 7470:7474 -p 8090:8090 -p 7680:7687 -p␣
↪6379:6379 docker-ckg:latest
```

Note: Be aware, this requires Docker to be previously installed.

CHAPTER

THREE

GETTING STARTED

- **Connecting to the CKG**: *Connect to DB*
- **Create a new user in the graph database**: *Create new user*
- **Create a project in the database**: *Project Creation*
- **Upload experimental data**: *Data Upload*
- **Define data analysis settings**: *Configuration*
- **Access the analysis report**: *Access report*
- **Report notification**: *Notifications*

3.1 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 one of Neo4j's Python drivers `Py2neo`, 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.

3.1.1 Py2neo 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 `py2neo` 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 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.

3.1.2 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.

3.2 Create a new user in the graph database

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 src/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 src/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.

3.3 Create a new project in the database

The project creation app in the Clinical Knowledge Graph was designed to make the process straightforward and user-friendly. To create a project, please follow the steps below.

Neo4j

1. Open neo4j desktop
2. Start the database

Terminal

1. In one terminal window:

- Activate the virtual environment (if created beforehand)

```
$ source /path/to/virtualenvironment/bin/activate;
```

- Start a redis-server:

```
$ redis-server
```

Warning: If redis-server is not found, install with `brew install redis` (Mac) or `sudo apt-get install redis-server` (Linux).

Warning: On Windows, redis-server should be installed by default (to check, go to start menu > services.msc > Redis). If that is not the case, go to `https://github.com/microsoftarchive/redis/releases`, download the latest release installer (.msi file), and follow the installation instructions. Let all options as default but remember to select **Add the Redis installation folder to the PATH environment variable.**. To start a redis-server, make sure it is **not running in ``services.msc``**, and run in the command prompt: C:"Program Files"Redisredis-server

Note: `C:\"Program Files"` can be replaced with the correct path where you installed Redis from the installer (.msi file).

2. In two separate terminal windows:

- Navigate to `report_manager` in both of them

```
$ cd CKG/src/report_manager
```

- Start a celery queue from the report_manager directory, in each window:

Default queue

```
$ celery -A worker worker --loglevel=DEBUG --concurrency=3 -E
```

In Windows, this corresponds to:

```
> celery worker -A worker --pool=eventlet --loglevel=DEBUG --concurrency=3 -E
```

Compute queue - Report generation

```
$ celery -A worker worker --loglevel=DEBUG --concurrency=3 -E -Q compute
```

To start this queue on Windows, please run:

```
> celery worker -A worker --pool=eventlet --loglevel=DEBUG --concurrency=3 -E -Q␣
↪compute
```

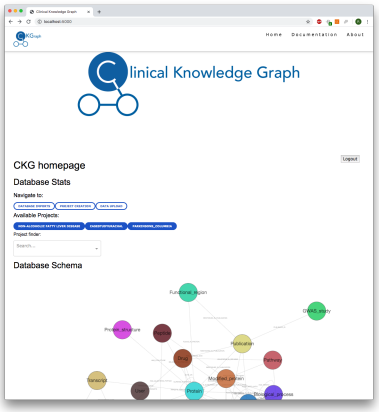
3. In a fourth terminal window:

- Run the report manager index app:

```
$ cd CKG/src/report_manager
$ python index.py
```

This will print some warnings, which should be okay.

Warning: Make sure that your virtual environment is always activated in each terminal window, before running any other command.



Browser

1. Copy the url `http://localhost:5000/` into a web browser and you will be directed to a login page.
2. Enter your username and password

This action will redirect you to the CKG home page app. From here, you can navigate to different applications, including the "Project Creation" app.

Note: Username and password will be authenticated in the CKG database. For this reason, you should have been created as a new user in the database before this step.

3.3.1 Project creation

From the CKG app home page, you can navigate to the project creation app by clicking `PROJECT CREATION` or pasting the url `http://localhost:5000/apps/projectCreationApp` in the browser.

Once you have been redirected, please fill in all the information needed to create a project. This includes all the fields marked with * (mandatory). **(1)** After all fields are filled in, please revise all the information and press `Create Project`. **(2)** The page will refresh and once finished, the project identifier will be depicted in front of the `Project information` header. **(3)** Use this identifier to search for data related to your project.

At this stage, and if your project has been successfully created in the database, a new button will appear and the message will instruct you to download a compressed file with the experimental design and clinical data template files. To do so, please press the button "Download Clinical Data template". **(4)**

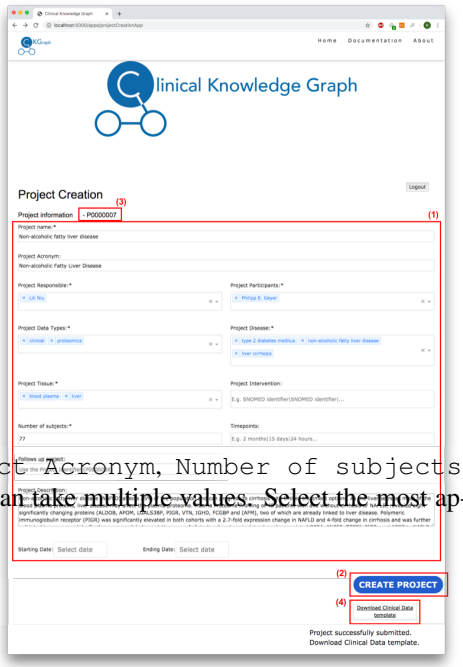


Fig. 1: Project Creation App

Note: Each field, with the exception of `Project name`, `Project Acronym`, `Number of subjects`, `Project Description`, `Starting Date` and `Ending Date`, can take multiple values. Select the most appropriate ones for your specific project.

Fill in the `ExperimentalDesign_Pxxxxxxx.xlsx` file with your subject, biological sample and analytical sample identifiers. Please double-check they are correct, this information is essential to map the results correctly in the database.

The `ClinicalData_Pxxxxxxx.xlsx` file needs to be filled in with all the relevant clinical data and sample information. For more instructions on how to fill in the file, please see *Upload project experimental data*.

To check your project in the neo4j database interface:

- Open the Neo4j desktop app
- Find the graph database in use and click *Manage*, followed by *Open Browser* (opens a new window).
- In the new Neo4j window, click on the database symbol (top left corner) and, under *Node Labels*, click *Project*

At this point, you should be able to see all the nodes corresponding to projects loaded in the database. To expand your project information, click on your project node and in the bottom of the window press the < symbol. Here you will find all the attributes of the project, including the project identifier (typically "P000000xx").

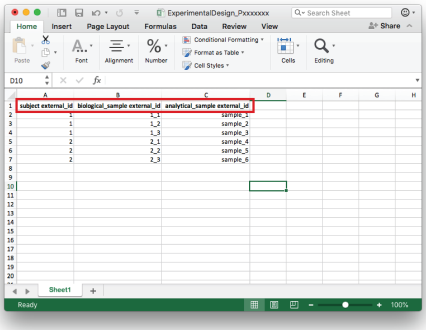
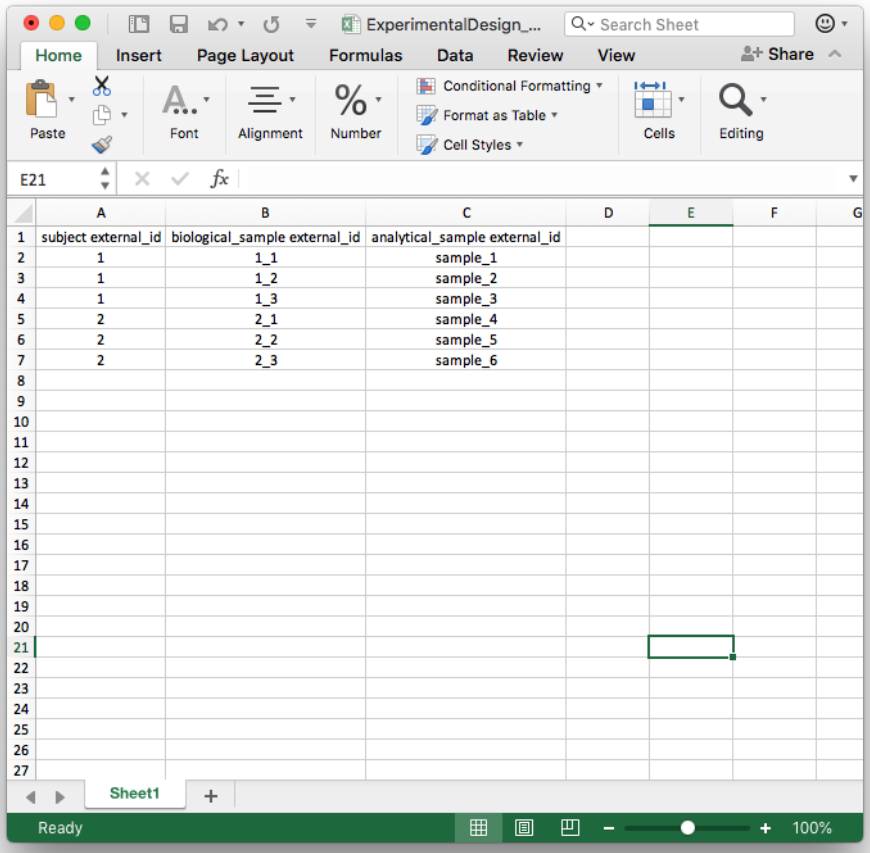


Fig. 2: Experimental Design file example

3.4 Upload project experimental data

3.4.1 Prepare data for upload

Experimental Design



	A	B	C
1	subject external_id	biological_sample external_id	analytical_sample external_id
2	1	1_1	sample_1
3	1	1_2	sample_2
4	1	1_3	sample_3
5	2	2_1	sample_4
6	2	2_2	sample_5
7	2	2_3	sample_6

Fig. 3: Experimental Design file example

Open the Experimental Design excel file, automatically downloaded when the project was created, and fill in the columns for *subject external_id*, *biological_sample external_id* and *analytical_sample external_id*. The identifiers provided in this file **must** correspond to the identifiers used in the *Clinical Data* file, and to the column names in the *Proteomics* files (see below).

Warning: Make sure, within each column, the identifiers are unique. This means, if you have a subject "KO1", no other subject can have the same identifier, but you can have a biological sample and/or analytical sample "KO1".

Clinical Data

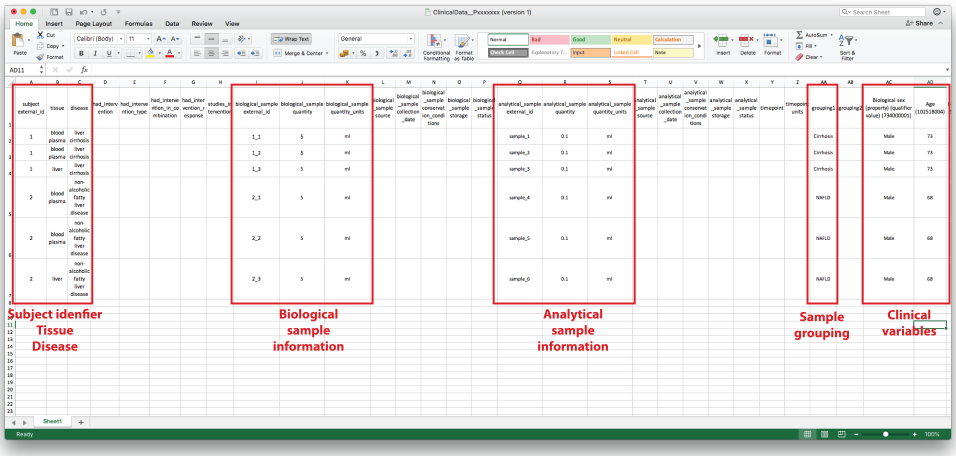


Fig. 4: Clinical Data file example

Open the Clinical Data excel file, automatically downloaded when the project was created, and fill in as much information as you can. Be aware that the following columns are mandatory to fill in:

- **subject external_id**: This is the identifier your subject has in your study so far (same identifiers as used in *Experimental Design*, **subject external_id**).
- **tissue**: This is the name of the tissue each sample came from. Make sure it is also one of the tissues selected during Project creation.
- **disease**: This should match the disease(s) you selected from the drop-down menu in the *Project creation*.
- **biological_sample external_id**: This is the identifier of the sample taken from your subject, if you have both blood and urine for every subject, you should correspondingly have two biological sample identifiers for each subject identifier (same identifiers as used in *Experimental Design*, **biological_sample external_id**).
- **biological_sample quantity**: Amount of biological sample.
- **biological_sample quantity_units**: Unit.
- **analytical_sample external_id**: If multiple analyses were performed on the same biological sample, eg. proteomics and transcriptomics, there should be multiple analytical sample identifiers for every biological sample (same identifiers as used in *Experimental Design*, **analytical_sample external_id**).
- **analytical_sample quantity**: Amount of sample used in the experiment.
- **analytical_sample quantity_units**: Unit.
- **grouping1**: Annotated grouping of each sample.
- **grouping2**: If there are more than one grouping (two independent variables) use this column to add a second level.

Additional clinical information about your study subjects can be added in the subsequent columns (i.e. columns after "grouping2"). Please use SNOMED terms as headers for every new column you add. This will be used to gather existing information about the type of data you have. To find an adequate SNOMED term for your clinical variables, please visit the SNOMED browser.

Note: Be aware, the two-independent-variable statistics is not yet implemented in the default analysis pipeline.

Note: To add a column with "Age" search for "age" in the SNOMED browser. This gives multiple matches, with the first one being: "Age (qualifier value), SCTID:397669002". Please enter this information as your clinical variable column header with the SCTID in parenthesis: Age (qualifier value) (397669002)

Warning: If an adequate SNOMED term is not available, please write an e-mail to annelaura.bach@cpr.ku.dk with the subject "Header Creation, CKG". In the email please provide your "missing" header and a description of what it is. Do this before uploading the Clinical Data.

Additional columns:

- **timpeoint**: To be used in the case of a longitudinal study. This is a relative measure within your samples timepoints. For example, if your timepoints are years 2015, 2016, 2017, 2018 and 2019, you would use "0", "1", "2", "3" and "4" as values in this column.
- **timepoint units**: Unit in which your **timepoint** is measured (e.g. "hours", "days", "years").
- **had_intervention**: If a subject has been subjected to a determined medical intervention. For now, select only drugs that have been given to the subject (e.g. "327032007"). Use an appropriate SNOMED SCTID value.
- **had_intervention_type**: This is the type of intervention applied to a subject. "drug treatment" is the only value available for now.
- **had_intervention_in_combination**: Boolean. If True, requires more than one value in **had_intervention**.
- **had_intervention_response**: "positive" or "negative".
- **studies_intervention**: A medical intervention under study in the project. For example, study subjects before and after stomach bypass (SCTID:442338001). Use an appropriate SNOMED SCTID value.

Proteomics data

- **MaxQuant**: Use "proteinGroups.txt", "peptides.txt" and "Oxidation (M)Sites.txt" files, and any other relevant MaxQuant output files.
- **Spectronaut**: Use "proteinGroupsReport.xlsx". When exporting the results table from Spectronaut, please select "PG.ProteinAccessions" and "PG.Qvalue" under *Row Labels*, and under *Cell Values* select "PG.Quantity", "PG.NrOfStrippedSequencesMeasured", "PG.NrOfStrippedSequencesIdentified", "PG.NrOfPrecursorsIdentified", "PG.IsSingleHit", "PG.NrOfStrippedSequencesUsedForQuantification", "PG.NrOfModifiedSequencesUsedForQuantification", "PG.NrOfPrecursorsUsedForQuantification", "PG.MS1Quantity" and "PG.MS2Quantity".

It is very important that all your column names have the following format: "LFQ intensity TechnicalReplicateNumber_AnalyticalSampleIdentifier" or "TechnicalReplicateNumber_AnalyticalSampleIdentifier.PG.Quantity". Where "TechnicalReplicateNumber_AnalyticalSampleIdentifier" should be replaced as shown in the example table below:

Technical replicate	Analytical sample id	Timepoint	Result
1	KO_plate1		1_KO_plate1
1	KO2_plate1	0	1_KO_plate1_0
1	KO3_plate1	30	1_KO_plate1_30
1	KO4_plate2		1_KO4_plate2
2	KO4_plate2		2_KO4_plate2

As shown in the example table, if your experimental design is a timecourse experiment, you should add "_" followed by the timepoint, right after the analytical sample identifier. Otherwise, you can omit it.

Do not perform any post-processing filtering, imputations or similar on your data before uploading it. This will be carried out by the CKG. In the case of Spectronaut outputs, the missing values are automatically replaced by the keyword "Filtered".

You can proceed to *Upload Data* when you have prepared your experimental design file, clinical and proteomics data.

3.4.2 Upload Data

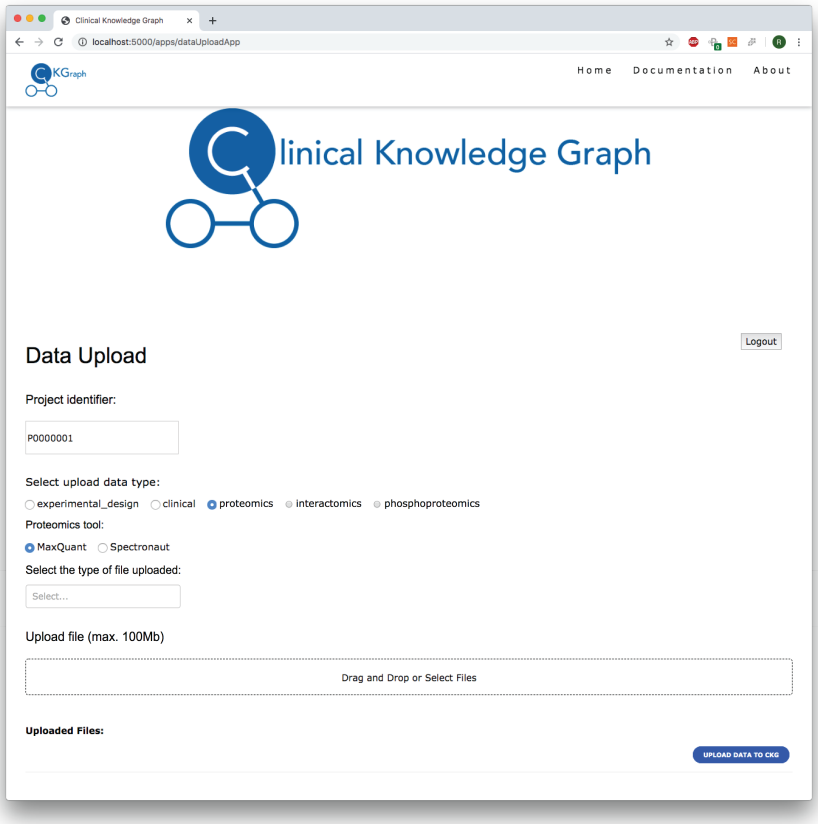


Fig. 5: Data Upload App

In order to make data uploading simple, we created an app that takes care of this in only a few steps:

Go to dataUploadApp or use the `Data Upload` button in the homepage app, and follow the steps.

1. Fill in `Project identifier` with your project external identifier from *Project creation* and press *Enter*. **(1)** If the project identifier does not exist in the database, you will get and error. Otherwise, the menus below will unlock.
2. Select the type of data you will upload first. **(2)**
    - If `proteomics`, `interactomics` or `phosphoproteomics` is selected, please also select the processing tool used (`MaxQuant` or `Spectronaut`) **(2a)**, as well as the type of file to be uploaded (`Protein groups`, `Peptides` or `Phospho STY sites`) **(2b)**.
3. Drag and drop or select the file to upload to the selected data type and file type. **(3)**

- If you want to upload, for example, both protein groups and peptides from a proteomics experiment, follow the steps **2.** and **3.** for each file type to be uploaded.

1. Select another data type to upload **(2)**, and drag and drop or select the files to upload **(3)**.
2. When you have uploaded all the relevant files, click `UPLOAD DATA TO CKG` **(4)**. After this button is clicked, it will deactivate all the menus. To restore its function, insert the project identifier and go through the previous steps again.
3. Once the data is uploaded, a new button will show under `UPLOAD DATA TO CKG`. Click `Download Uploaded Files (.zip)` to download all the upload files in a compressed format.

Note: When the files are uploaded, the filenames are shown under `Uploaded Files`: To replace the files uploaded, just select the correct data type and processing tool, and reselect the files again.

3.5 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 default workflow makes use of these resources and runs, for each data type, the analysis pipeline defined in a configuration file. In the CKG, we have default analysis defined for Clinical data, Proteomics, and Multiomics. All the analysis configuration files can be modified to fit your project or data.

To check how each configuration file looks like and how to modify them, please follow the links below.

3.5.1 Clinical data analysis parameters

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:

- **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

Fig. 6: Clinical Data configuration file

- **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:

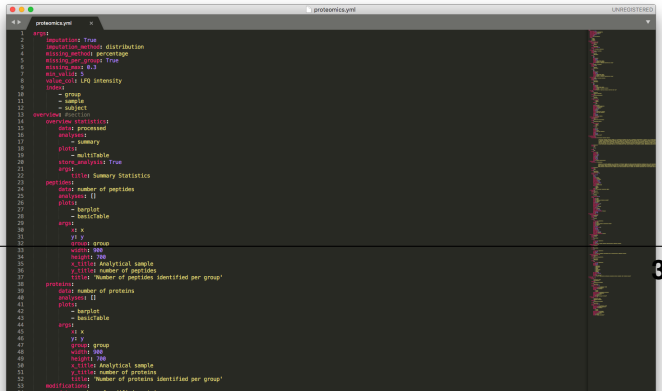
- **description**: Definition of the analysis used.
- **data**: defines on which dataset dataframe the analysis will be ran (e.g. "clinical variables", "original", "processed").
- **analyses**: which statistical analysis to run on the `data`. These functions are called from the mocdule `analytics_factory.py`.
- **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. If you would like to add a specific analysis step to the default pipeline, remember to add a call to the function in `analytics_factory.py`.

3.5.2 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 `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.

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. If you would like to add a specific analysis step to the default pipeline, remember to add a call to the function in `analytics_factory.py`.

3.5.3 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, and a Weighted Gene Co-expression Network Analysis (WGCNA) section, where features (proteins) are clustered in co-expression modules and further correlated to the clinical variables.

3.6 Accessing the analysis report

The standard report provides initial evaluation of the quality of the generated data, highlight relevant hits, and contextualize these hits in relation to different biomedical components in the graph.

There are two ways to acces the report, on the browser using a Dash App, or in a Jupyter notebook. For more information on each, please follow the links below.

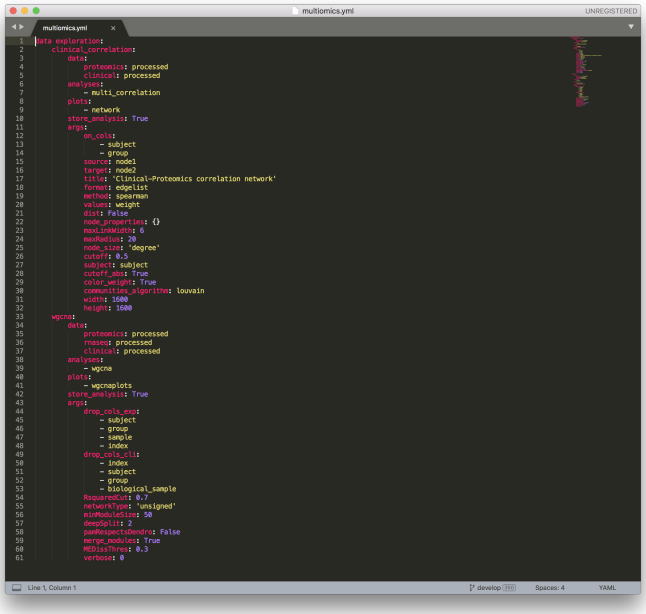


Fig. 9: Multiomics configuration file

3.6.1 Report: Dash web app

3.6.2 Report: Jupyter-notebook

- Easy access to all the experimental data generated in the lab
- All the results from statistical analysis
- Visualization of reports
- All the python functionality at hand
- Additional analysis of project (e.g. treatment candidate selection, PTMs, predictions)

3.7 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,

3.7.1 Slack notifications

3.7.2 E-mail notifications

CHAPTER

FOUR

THE PROJECT REPORT

- **Generate a project**: *Project*
- **The Tabs**: *Project tabs*

4.1 Generating a project report

4.2 Project report tabs

CHAPTER

FIVE

CKG BUILDER

- **Ontology sources and parsers**: *Ontologies*
- **Biomedical databases and resources**: *Databases*
- **Parsing experimental data**: *Experiments*
- **Building the graph database from one module**: *Builder*

5.1 Ontology sources and raw file parsers

5.2 Biomedical databases and resources

5.3 Parsing experimental data

5.3.1 Clinical data

5.3.2 Proteomics

- MaxQuant
- Biognosys

5.3.3 Phosphoproteomics

5.3.4 WES

5.4 Building the graph database from one Python module

CHAPTER

SIX

ADVANCED FEATURES

- **CKG Statistics**: *Imports stats* | *Graph database stats*
- **Jupyter notebooks**: *Notebooks*
- **Retrieving data from the CKG**: *DB Querying*
- **Data Analysis**: *Analysis*
- **Visualization**: *Plots*
- **R interface**: *R wrapper*

6.1 Clinical Knowledge Graph Statistics: Imports

6.2 Clinical Knowledge Graph Statistics: Database

6.3 Using Jupyter Notebooks with the Clinical Knowledge Graph

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 `src/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.

6.3.1 Reporting notebooks

Reporting notebooks refers to Jupyter notebooks that have been finished and properly tested by the developers, and are ready to be used by the community.

- **project_reporting.ipynb**

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.ipynb**

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.ipynb**

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 loading all 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) controls within the notebook. In this case, the user can select different clusters of proteins (from the correlation network) and observe their variation across groups.

- **Urachal Carcinoma Case Study.ipynb**

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 filter for 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 down the 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 the studied disease, further reducing the list of potential drugs candidates.

6.3.2 Development notebooks

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

When a notebook in this folder is functional and successfully benchmarked, the notebook is moved to the reporting directory.

6.4 Retrieving data from the Clinical Knowledge Graph database

6.5 Standardising the data analysis

6.6 Visualisation plots

6.7 Python - R interface

CHAPTER

SEVEN

SYSTEM REQUIREMENTS

7.1 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.

7.1.1 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*.

7.1.2 Software

- Java, version compatible with chosen Neo4j version (check Neo4j requirements)
- R >= 3.5.2
- Python 3.6
- Redis server
- Neo4j Desktop
- Neo4j database =< 3.14

7.1.3 Hardware

- Memory: 16Gb
- Disk space: >= 80Gb
- Stable internet connection

CHAPTER

EIGHT

API REFERENCE

8.1 CKG package API Reference

graphdb_connector
graphdb_builder
report_manager
analytics_core
notebooks

8.1.1 Graph Database Connector

connector.py

graphdb_connector.connector.**getGraphDatabaseConnectionConfiguration**(*configuration=None*, *database=None*)

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

graphdb_connector.connector.**removeRelationshipDB**(*entity1*, *entity2*, *relationship*)

graphdb_connector.connector.**modifyEntityProperty**(*parameters*)
parameters: tuple with entity name, entity id, property name to modify, and value

graphdb_connector.connector.**sendQuery**(*driver*, *query*, *parameters={}*)

graphdb_connector.connector.**getCursorData**(*driver*, *query*, *parameters={}*)

graphdb_connector.connector.**create_node**(*driver*, *node_type*, ***kwargs*)

graphdb_connector.connector.**find_node**(*driver*, *node_type*, ***kwargs*)

query_utils.py

graphdb_connector.query_utils.**read_knowledge_queries**(*dataset_type='proteomics'*)

graphdb_connector.query_utils.**read_queries**(*queries_file*)

graphdb_connector.query_utils.**list_queries**(*queries*)

graphdb_connector.query_utils.**find_queries_involving_nodes**(*queries*, *nodes*, *print_pretty=False*)

graphdb_connector.query_utils.**find_queries_involving_relationships**(*queries*, *rels*)

graphdb_connector.query_utils.**get_query**(*queries*, *query_id*)

graphdb_connector.query_utils.**get_description**(*query*)

graphdb_connector.query_utils.**get_nodes**(*query*)

graphdb_connector.query_utils.**get_relationships**(*query*)

graphdb_connector.query_utils.**map_node_name_to_id**(*driver*, *node*, *value*)

8.1.2 Graph Database Builder

Ontology Databases

Ontologies Parsers

icdParser.py

graphdb_builder.ontologies.parsers.icdParser.**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

graphdb_builder.ontologies.parsers.oboParser.**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

graphdb_builder.ontologies.parsers.reflectParser.**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

graphdb_builder.ontologies.parsers.snomedParser.**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*).

graphdb_builder.ontologies.parsers.snomedParser.**get_inactive_terms**(*concept_file*)

Parameters **concept_file** –

Return set inactive_terms inactive terms

Ontologies Controller

graphdb_builder.ontologies.ontologies_controller.**entries_to_remove**(*entries*, *the_dict*)

This function removes pairs from a given dictionary, based on a list of provided keys.

Parameters

- **entries** (*list*) – list of keys to be deleted from dictionary.
- **the_dict** (*dict*) – dictionary.

Returns The original dictionary minus the key,value pairs from the provided entries list.

graphdb_builder.ontologies.ontologies_controller.**get_extra_entities_rels**(*ontology_directory*)

graphdb_builder.ontologies.ontologies_controller.**parse_ontology**(*ontology*, *download=True*)

Parses and extracts data from a given ontology file(s), and returns a tuple with multiple dictionaries.

Parameters

- **ontology** (*str*) – acronym of the ontology to be parsed (e.g. Disease Ontology:'DO').
- **download** (*bool*) – wether database is to be downloaded.

Returns Tuple with three nested dictionaries: terms, relationships between terms, and definitions of the terms. For more information on the returned dictionaries, see the documentation for any ontology parser.

graphdb_builder.ontologies.ontologies_controller.**generate_graphFiles**(*import_directory*, *ontologies=None*, *download=True*)

This function parses and extracts data from a given list of ontologies. If no ontologies are provided, all availables ontologies are used. Terms, relationships and definitions are saved as .tsv files to be loaded into the graph database.

Parameters

- **import_directory** (*str*) – relative path from current python module to 'imports' directory.
- **ontologies** (*list or None*) – list of ontologies to be imported. If None, all available ontologies are imported.
- **download** (*bool*) – wether database is to be downloaded.

Returns Dictionary of tuples. Each tuple corresponds to a unique label/relationship type, date, time, database, and number of nodes and relationships.

Biomedical Databases

Biomedical Databases Parsers

cancerGenomeInterpreterParser.py

graphdb_builder.databases.parsers.cancerGenomeInterpreterParser.**parser**(*databases_directory*, *download=True*)

corumParser.py

graphdb_builder.databases.parsers.corumParser.**parser**(*databases_directory*, *download=True*)

disgenetParser.py

graphdb_builder.databases.parsers.disgenetParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.disgenetParser.**readDisGeNetProteinMapping**(*config*, *directory*)

graphdb_builder.databases.parsers.disgenetParser.**readDisGeNetDiseaseMapping**(*config*, *directory*)

drugBankParser.py

graphdb_builder.databases.parsers.drugBankParser.**parser**(*databases_directory*)

graphdb_builder.databases.parsers.drugBankParser.**extract_drugs**(*config*, *directory*)

graphdb_builder.databases.parsers.drugBankParser.**parseDrugBankVocabulary**(*config*, *directory*)

graphdb_builder.databases.parsers.drugBankParser.**build_relationships_from_DrugBank**(*config*, *drugs*)

graphdb_builder.databases.parsers.drugBankParser.**build_drug_entity**(*config*, *drugs*)

graphdb_builder.databases.parsers.drugBankParser.**build_DrugBank_dictionary**(*config*, *directory*, *drugs*)

drugGeneInteractionDBParser.py

graphdb_builder.databases.parsers.drugGeneInteractionDBParser.**parser**(*databases_directory*, *download=True*)

exposomeParser.py

graphdb_builder.databases.parsers.exposomeParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.exposomeParser.**parseBiomarkersFile**(*fhandler*, *file_name*)

graphdb_builder.databases.parsers.exposomeParser.**parseCorrelationsFile**(*fhandler*, *file_name*, *biomarkers*, *mapping*)

foodbParser.py

graphdb_builder.databases.parsers.foodbParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.foodbParser.**parseContents**(*fhandler*)

graphdb_builder.databases.parsers.foodbParser.**parseFood**(*fhandler*)

graphdb_builder.databases.parsers.foodbParser.**parseCompounds**(*fhandler*)

goaParser.py

graphdb_builder.databases.parsers.goaParser.**parser**(*databases_dir*, *download=True*)

graphdb_builder.databases.parsers.goaParser.**parse_annotations_with_pandas**(*annotation_file*, *valid_proteins=None*)

gwasCatalogParser.py

graphdb_builder.databases.parsers.gwasCatalogParser.**parser**(*databases_directory*, *download=True*)

hgncParser.py

graphdb_builder.databases.parsers.hgncParser.**parser**(*databases_directory*, *download=True*)

hmdbParser.py

graphdb_builder.databases.parsers.hmdbParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.hmdbParser.**extract_metabolites**(*config*, *directory*, *download=True*)

graphdb_builder.databases.parsers.hmdbParser.**build_metabolite_entity**(*config*, *directory*, *metabolites*)

graphdb_builder.databases.parsers.hmdbParser.**build_relationships_from_HMDB**(*config*, *metabolites*, *mapping*)

graphdb_builder.databases.parsers.hmdbParser.**build_HMDB_dictionary**(*directory*, *metabolites*)

hpaParser.py

graphdb_builder.databases.parsers.hpaParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.hpaParser.**parsePathologyFile**(*config*, *fhandler*, *file_name*, *protein_mapping*, *disease_mapping*)

intactParser.py

graphdb_builder.databases.parsers.intactParser.**parser**(*databases_directory*, *download=True*)

internalDBsParser.py

graphdb_builder.databases.parsers.jensenlabParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.jensenlabParser.**parsePairs**(*config*, *databases_directory*, *qtype*, *mapping*, *download=True*)

mutationDsParser.py

graphdb_builder.databases.parsers.mutationDsParser.**parser**(*databases_directory*, *download=True*)

oncokbParser.py

graphdb_builder.databases.parsers.oncokbParser.**parser**(*databases_directory*, *download=True*)

pathwayCommonsParser.py

graphdb_builder.databases.parsers.pathwayCommonsParser.**parser**(*databases_directory*, *download=True*)

pfamParser.py

graphdb_builder.databases.parsers.pfamParser.**parser**(*databases_directory*, *import_directory*, *download=True*, *updated_on=None*)

graphdb_builder.databases.parsers.pfamParser.**print_files**(*data*, *header*, *outputfile*, *is_first*, *filter_for=None*)

pspParser.py

graphdb_builder.databases.parsers.pspParser.**parser**(*databases_directory*)

graphdb_builder.databases.parsers.pspParser.**parseSites**(*fhandler*, *modifications*)

graphdb_builder.databases.parsers.pspParser.**parseKinaseSubstrates**(*fhandler*, *modifications*)

graphdb_builder.databases.parsers.pspParser.**parseRegulationAnnotations**(*fhandler*, *modifications*, *mapping*)

graphdb_builder.databases.parsers.pspParser.**parseDiseaseAnnotations**(*fhandler*, *modifications*, *mapping*)

reactomeParser.py

graphdb_builder.databases.parsers.reactomeParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.reactomeParser.**parsePathways**(*config*, *databases_directory*, *fhandler*)

graphdb_builder.databases.parsers.reactomeParser.**parsePathwayHierarchy**(*fhandler*)

graphdb_builder.databases.parsers.reactomeParser.**parsePathwayRelationships**(*config*, *fhandler*, *mapping=None*)

refseqParser.py

graphdb_builder.databases.parsers.refseqParser.**parser**(*databases_directory*, *download=True*)

siderParser.py

graphdb_builder.databases.parsers.siderParser.**parser**(*databases_directory*, *drug_source*, *download=True*)

graphdb_builder.databases.parsers.siderParser.**parserIndications**(*databases_directory*, *drugMapping*, *phenotypeMapping*, *download=True*)

smpdbParser.py

graphdb_builder.databases.parsers.smpdbParser.**parser**(*databases_directory*, *download=True*)

graphdb_builder.databases.parsers.smpdbParser.**parsePathways**(*config*, *fhandler*)

graphdb_builder.databases.parsers.smpdbParser.**parsePathwayProteinRelationships**(*fhandler*)

graphdb_builder.databases.parsers.smpdbParser.**parsePathwayMetaboliteDrugRelationships**(*fhandl*

stringParser.py

graphdb_builder.databases.parsers.stringParser.**parser**(*databases_directory*, *importDirectory*, *drug_source=None*, *download=True*, *db='STRING'*)

graphdb_builder.databases.parsers.stringParser.**parseActions**(*databases_directory*, *importDirectory*, *proteinMapping*, *drugMapping=None*, *download=True*, *db='STRING'*)

textminingParser.py

graphdb_builder.databases.parsers.textminingParser.**parser**(*databases_directory*, *importDirectory*, *download=True*)

graphdb_builder.databases.parsers.textminingParser.**read_valid_pubs**(*organisms*, *organisms_file*)

graphdb_builder.databases.parsers.textminingParser.**parse_PMC_list**(*config*, *directory*, *download=True*, *valid_pubs=None*)

graphdb_builder.databases.parsers.textminingParser.**parse_mentions**(*config*, *directory*, *qtype*, *importDirectory*, *download=True*)

uniprotParser.py

graphdb_builder.databases.parsers.uniprotParser.**parser**(*databases_directory*, *import_directory*, *download=True*, *updated_on=None*)

graphdb_builder.databases.parsers.uniprotParser.**parse_release_notes**(*databases_directory*, *config*, *download=True*)

graphdb_builder.databases.parsers.uniprotParser.**parse_fasta**(*databases_directory*, *config*, *import_directory*, *download=True*, *updated_on=None*)

graphdb_builder.databases.parsers.uniprotParser.**parse_idmapping_file**(*databases_directory*, *config*, *import_directory*, *download=True*, *updated_on=None*)

graphdb_builder.databases.parsers.uniprotParser.**format_output**(*proteins*)

graphdb_builder.databases.parsers.uniprotParser.**print_single_file**(*data*, *header*, *output_file*, *data_type*, *data_object*, *is_first*, *updated_on*)

graphdb_builder.databases.parsers.uniprotParser.**print_multiple_relationships_files**(*data*, *header*, *output_dir*, *is_first*, *updated_on*)

graphdb_builder.databases.parsers.uniprotParser.**addUniProtTexts**(*textsFile*, *proteins*)

graphdb_builder.databases.parsers.uniprotParser.**parseUniProtVariants**(*config*, *databases_directory*, *import_directory*, *download=True*, *updated_on=None*)

graphdb_builder.databases.parsers.uniprotParser.**parseUniProtAnnotations**(*config*, *databases_directory*, *download=True*)

graphdb_builder.databases.parsers.uniprotParser.**parseUniProtPeptides**(*config*, *databases_directory*, *download=True*)

Databases Controller

graphdb_builder.databases.databases_controller.**parseDatabase**(*importDirectory*, *database*, *download=True*)

graphdb_builder.databases.databases_controller.**generateGraphFiles**(*importDirectory*, *databases=None*, *download=True*, *n_jobs=4*)

Experimental Data

Experiments Parsers

Clinical parser

graphdb_builder.experiments.parsers.clinicalParser.**parser**(*projectId*)

graphdb_builder.experiments.parsers.clinicalParser.**project_parser**(*projectId*, *config*, *directory*, *separator*)

graphdb_builder.experiments.parsers.clinicalParser.**experimental_design_parser**(*projectId*, *config*, *directory*)

graphdb_builder.experiments.parsers.clinicalParser.**clinical_parser**(*projectId*, *config*, *directory*, *separator*)

graphdb_builder.experiments.parsers.clinicalParser.**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

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_info**(*project_data*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_responsible_rels**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_participant_rels**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_tissue_rels**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_disease_rels**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_intervention_rels**(*project_*
separator='|')

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_rels**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_timepoints**(*project_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_project_subject_rels**(*projectId*, *design_data*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_subject_identifiers**(*design_data*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_biosample_identifiers**(*design_data*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_analytical_sample_identifiers**(*de*

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_subject_rels**(*d*

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_analytical_sa**

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_samples_info**(*clinical_da*

graphdb_builder.experiments.parsers.clinicalParser.**extract_analytical_samples_info**(*clinical_da*

graphdb_builder.experiments.parsers.clinicalParser.**extract_biosample_analytical_sample_rela**

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_timepoint_rels**

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_tissue_rels**(*cl*

graphdb_builder.experiments.parsers.clinicalParser.**extract_subject_disease_rels**(*clinical_data*, *separator='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_subject_intervention_rels**(*clinical_sep-a-ra-tor='|'*)

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_group_rels**(*clin*

graphdb_builder.experiments.parsers.clinicalParser.**extract_biological_sample_clinical_varia**

Proteomics parser

graphdb_builder.experiments.parsers.proteomicsParser.**parser**(*projectId*, *directory=None*)

graphdb_builder.experiments.parsers.proteomicsParser.**parse_from_directory**(*projectId*, *directory*, *configuration=None*)

graphdb_builder.experiments.parsers.proteomicsParser.**parser_from_file**(*file_path*, *configuration*, *data_type*, *is_standard=True*)

graphdb_builder.experiments.parsers.proteomicsParser.**get_configuration**(*processing_tool*, *data_type*)

graphdb_builder.experiments.parsers.proteomicsParser.**update_configuration**(*data_type*, *processing_tool*, *value_col='LFQ intensity'*, *columns=[]*, *drop_cols=[]*, *filters=None*, *new_config={}*)

graphdb_builder.experiments.parsers.proteomicsParser.**parse_dataset**(*filepath*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**parse_standard_dataset**(*file_path*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**check_columns**(*data*, *req_columns*, *generated_columns*)

graphdb_builder.experiments.parsers.proteomicsParser.**check_minimum_configuration**(*configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**load_dataset**(*uri*, *configuration*)

This function gets the molecular data from a proteomics experiment. Input: uri of the processed file resulting from MQ Output: pandas DataFrame with the columns and filters defined in config.py

graphdb_builder.experiments.parsers.proteomicsParser.**remove_contaminant_tag**(*column*, *tag='CON__'*)

graphdb_builder.experiments.parsers.proteomicsParser.**expand_groups**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_modification_protein_rels**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_protein_modification_subject_**

graphdb_builder.experiments.parsers.proteomicsParser.**extract_protein_protein_modification_**

graphdb_builder.experiments.parsers.proteomicsParser.**extract_peptide_protein_modification_**

graphdb_builder.experiments.parsers.proteomicsParser.**extract_protein_modifications_rels**(*data*, *configuration*

graphdb_builder.experiments.parsers.proteomicsParser.**extract_protein_modifications_modifica**

graphdb_builder.experiments.parsers.proteomicsParser.**extract_peptides**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_peptide_subject_rels**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_peptide_protein_rels**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_protein_subject_rels**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**get_value_cols**(*data*, *configuration*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_subject_replicates_from_regex**

graphdb_builder.experiments.parsers.proteomicsParser.**extract_subject_replicates**(*data*, *value_cols*)

graphdb_builder.experiments.parsers.proteomicsParser.**extract_attributes**(*data*, *attributes*)

graphdb_builder.experiments.parsers.proteomicsParser.**merge_regex_attributes**(*data*, *attributes*, *index*, *regexCols*)

graphdb_builder.experiments.parsers.proteomicsParser.**merge_col_attributes**(*data*, *attributes*, *index*)

graphdb_builder.experiments.parsers.proteomicsParser.**calculate_median_replicates**(*data*, *log='log2'*)

graphdb_builder.experiments.parsers.proteomicsParser.**update_groups**(*data*, *groups*)

graphdb_builder.experiments.parsers.proteomicsParser.**get_dataset_configuration**(*processing_forma*, *data_type*)

WES parser

graphdb_builder.experiments.parsers.wesParser.**parser**(*projectId*)

graphdb_builder.experiments.parsers.wesParser.**parseWESDataset**(*projectId*, *configuration*, *dataDir*)

graphdb_builder.experiments.parsers.wesParser.**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

graphdb_builder.experiments.parsers.wesParser.**extractWESRelationships**(*data*, *configuration*)

Experiments Controller

graphdb_builder.experiments.experiments_controller.**generate_dataset_imports**(*projectId*, *dataType*, *dataset_import_dir*)

graphdb_builder.experiments.experiments_controller.**generate_graph_files**(*data*, *dataType*, *projectId*, *stats*, *ot='w'*, *dataset_import_dir='experi*

User Creation

Users Controller

graphdb_builder.users.users_controller.**parseUsersFile**(*importDirectory*, *expiration=365*)

Creates new user in the graph database and corresponding node, through the following steps:

1. Generates new user identifier
2. Checks if a user with given properties already exists in the database. If not:
3. Creates new local user (access to graph database)
4. Saves data to tab-delimited file.

Parameters

- **importDirectory** (*str*) – path to the directory where all the import files are generated.
- **expiration** (*int*) – number of days a user is given access.

Returns Writes relevant .tsv file for the users in the provided file.

graphdb_builder.users.users_controller.**get_user_creation_queries**()
Reads the YAML file containing the queries relevant to user creation, parses the given stream and returns a Python object (dict[dict]).

graphdb_builder.users.users_controller.**get_new_user_identifier**(*driver*)
Queries the database for the last user identifier and returns a new sequential identifier.

Parameters **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.

Returns User identifier.

Return type str

graphdb_builder.users.users_controller.**check_if_node_exists**(*driver*, *node_property*, *value*)
Queries the graph database and checks if a node with a specific property and property value already exists.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **node_property** (*str*) – property of the node.
- **value** (*str, int, float or bool*) – property value.

Returns Pandas dataframe with user identifier if User with node_property and value already exists, if User does not exist, returns and empty dataframe.

graphdb_builder.users.users_controller.**create_db_user**(*driver*, *data*)
Creates and assigns role to new graph database user, if user not in list of local users.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **data** (*Series*) – pandas Series with required user information (see set_arguments()).

graphdb_builder.users.users_controller.**GenerateGraphFiles**(*data*, *output_file*)
Saves pandas dataframe to users.tsv. If file already exists, appends new lines. Else, creates file and writes dataframe to it.

Parameters

- **data** – pandas dataframe to be written to .tsv file.
- **output_file** (*str*) – path to output csv file.

CKG Builder

User Creation Module

graphdb_builder.builder.create_user.**create_user_node**(*driver*, *data*)

Creates graph database node for new user and adds respective properties to node.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **data** (*Series*) – pandas Series with new user identifier and required user information (see set_arguments()).

graphdb_builder.builder.create_user.**create_user_from_command_line**(*args*, *expiration*)

Creates new user in the graph database and corresponding node, from a terminal window (command line), and adds the new user information to the users excel and import files. Arguments as in set_arguments().

Parameters

- **args** (*any object with __dict__ attribute*) – object. Contains all the parameters neccessary to create a user ('username', 'name', 'email', 'secondary_email', 'phone_number' and 'affiliation').
- **expiration** (*int*) – number of days users is given access.

Note: This function can be used directly with *python create_user_from_command_line.py -u username -n user_name -e email -s secondary_email -p phone_number -a affiliation .*

---

graphdb_builder.builder.create_user.**create_user_from_file**(*filepath*, *expiration*)

Creates new user in the graph database and corresponding node, from an excel file. Rows in the file must be users, and columns must follow set_arguments() fields.

Parameters

- **filepath** (*str*) – filepath and filename containing users information.
- **output_file** (*str*) – path to output csv file.
- **expiration** (*int*) – number of days users is given access.

Note: This function can be used directly with *python create_user_from_file.py -f path_to_file .*

---

graphdb_builder.builder.create_user.**create_user**(*data*, *output_file*, *expiration=365*)

Creates new user in the graph database and corresponding node, through the following steps:

1. Checks if a user with given properties already exists in the database. If not:
2. Generates new user identifier
3. Creates new local user (access to graph database)
4. Creates new user node
5. Saves data to users.tsv

Parameters

- **data** – pandas dataframe with users as rows and arguments and columns.
- **output_file** (*str*) – path to output csv file.
- **expiration** (*int*) – number of days users is given access.

Returns Writes relevant .tsv file for the users in data.

graphdb_builder.builder.create_user.**set_arguments**()
This function sets the arguments to be used as input for **create_user.py** in the command line.

Importer Module

Generates all the import files: Ontologies, Databases and Experiments. The module is reponsible for generating all the csv files that will be loaded into the Graph database and also updates a stats object (hdf table) with the number of entities and relationships from each dataset imported. A new stats object is created the first time a full import is run.

graphdb_builder.builder.importer.**ontologiesImport**(*importDirectory*, *ontologies=None*, *download=True*, *import_type='partial'*)
Generates all the entities and relationships from the provided ontologies. If the ontologies list is not provided, then all the ontologies listed in the configuration will be imported (full_import). This function also updates the stats object with numbers from the imported ontologies.

Parameters

- **importDirectory** (*str*) – path of the import directory where files will be created.
- **ontologies** (*list*) – a list of ontology names to be imported.
- **download** (*bool*) – wether database is to be downloaded.
- **import_type** (*str*) – type of import (´full´ or ´partial´).

graphdb_builder.builder.importer.**databasesImport**(*importDirectory*, *databases=None*, *n_jobs=1*, *download=True*, *import_type='partial'*)
Generates all the entities and relationships from the provided databases. If the databases list is not provided, then all the databases listed in the configuration will be imported (full_import). This function also updates the stats object with numbers from the imported databases.

Parameters

- **importDirectory** (*str*) – path of the import directory where files will be created.
- **databases** (*list*) – a list of database names to be imported.
- **n_jobs** (*int*) – number of jobs to run in parallel. 1 by default when updating one database.
- **import_type** (*str*) – type of import (´full´ or ´partial´).

graphdb_builder.builder.importer.**experimentsImport**(*projects=None*, *n_jobs=1*, *import_type='partial'*)
Generates all the entities and relationships from the specified Projects. If the projects list is not provided, then all the projects the experiments directory will be imported (full_import). Calls function experimentImport.

Parameters

- **projects** (*list*) – list of project identifiers to be imported.
- **n_jobs** (*int*) – number of jobs to run in parallel. 1 by default when updating one project.
- **import_type** (*str*) – type of import (´full´ or ´partial´).

graphdb_builder.builder.importer.**experimentImport**(*importDirectory*, *experimentsDirectory*, *project*)
Generates all the entities and relationships from the specified Project. Called from function experimentsImport.

Parameters

- **importDirectory** (*str*) – path to the directory where all the import files are generated.
- **experimentDirectory** (*str*) – path to the directory where all the experiments are located.
- **project** (*str*) – identifier of the project to be imported.

graphdb_builder.builder.importer.**usersImport**(*importDirectory*, *import_type='partial'*)
Generates User entities from excel file and grants access of new users to the database. This function also writes the relevant information to a tab-delimited file in the import directory.

Parameters

- **importDirectory** (*str*) – path to the directory where all the import files are generated.
- **import_type** (*str*) – type of import (´full´ or ´partial).

graphdb_builder.builder.importer.**fullImport**(*download=True*, *n_jobs=4*)
Calls the different importer functions: Ontologies, databases, experiments. The first step is to check if the stats object exists and create it otherwise. Calls setupStats.

graphdb_builder.builder.importer.**generateStatsDataFrame**(*stats*)
Generates a dataframe with the stats from each import. :param list stats: a list with statistics collected from each importer function. :return: Pandas dataframe with the collected statistics.

graphdb_builder.builder.importer.**setupStats**(*import_type*)
Creates a stats object that will collect all the statistics collected from each import.

graphdb_builder.builder.importer.**createEmptyStats**(*statsCols*, *statsFile*, *statsName*)
Creates a HDFStore object with a empty dataframe with the collected stats columns.

Parameters

- **statsCols** (*list*) – a list of columns with the fields collected from the import statistics.
- **statsFile** (*str*) – path where the object should be stored.
- **statsName** (*str*) – name if the file containing the stats object.

graphdb_builder.builder.importer.**writeStats**(*statsDf*, *import_type*, *stats_name=None*)
Appends the new collected statistics to the existing stats object. :param statsDf: a pandas dataframe with the new statistics from the importing. :param str statsName: If the statistics should be stored with a specific name.

graphdb_builder.builder.importer.**getStatsName**(*import_type*)
Generates the stats object name where to store the importing statistics from the CKG version, which is defined in the configuration.

Returns statsName: key used to store in the stats object.

Return type str

Loader Module

Populates the graph database with all the files generated by the importer.py module: Ontologies, Databases and Experiments. The module loads all the entities and relationships defined in the importer files. It calls Cypher queries defined in the cypher.py module. Further, it generates an hdf object with the number of enities and relationships loaded for each Database, Ontology and Experiment. This module also generates a compressed backup file of all the loaded files.

There are two types of updates:

- Full: all the entities and relationships in the graph database are populated
- Partial: only the specified entities and relationships are loaded

The compressed files for each type of update are named accordingly and saved in the archive/ folder in data/.

graphdb_builder.builder.loader.**load_into_database**(*driver*, *queries*, *requester*)
This function runs the queries provided in the graph database using a py2neo driver.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **queries** (*list[dict]*) – list of queries to be passed to the database.
- **requester** (*str*) – identifier of the query.

graphdb_builder.builder.loader.**updateDB**(*driver*, *imports=None*, *specific=[]*)
Populates the graph database with information for each Database, Ontology or Experiment specified in imports. If imports is not defined, the function populates the entire graph database based on the graph variable defined in the grapher_config.py module. This function also updates the graph stats object with numbers from the loaded entities and relationships.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **imports** (*list*) – a list of entities to be loaded into the graph.

graphdb_builder.builder.loader.**fullUpdate**()
Main method that controls the population of the graph database. Firstly, it gets a connection to the database (driver) and then initiates the update of the entire database getting all the graph entities to update from configuration. Once the graph database has been populated, the imports folder in data/ is compressed and archived in the archive/ folder so that a backup of the imports files is kept (full).

graphdb_builder.builder.loader.**partialUpdate**(*imports*, *specific=[]*)
Method that controls the update of the graph database with the specified entities and relationships. Firstly, it gets a connection to the database (driver) and then initiates the update of the specified graph entities. Once the graph database has been populated, the data files uploaded to the graph are compressed and archived in the archive/ folder (partial).

Parameters **imports** (*list*) – list of entities to update

graphdb_builder.builder.loader.**archiveImportDirectory**(*archive_type='full'*)
This function creates the compressed backup imports folder with either the whole folder (full update) or with only the files uploaded (partial update). The folder or files are compressed into a gzipped tarball file and stored in the archive/ folder defined in the configuration.

Parameters **archive_type** (*str*) – whether it is a full update or a partial update.

Builder Module

Builds the database in two main steps:

1) Imports all the data from ontologies, databases and experiments
2) Loads these data into the database

The module can perform full updates, executing both steps for all the ontologies, databases and experiments or a partial update. Partial updates can execute step 1 or step 2 for specific data.

graphdb_builder.builder.builder.**set_arguments**()
This function sets the arguments to be used as input for **builder.py** in the command line.

builder_utils.py

graphdb_builder.builder_utils.**readDataset**(*uri*)

graphdb_builder.builder_utils.**readDataFromCSV**(*uri*)
Read the data from csv file

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

graphdb_builder.builder_utils.**readDataFromExcel**(*uri*)
Read the data from Excel file

graphdb_builder.builder_utils.**get_files_by_pattern**(*regex_path*)

graphdb_builder.builder_utils.**get_extra_pairs**(*directory*, *extra_file*)

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

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

graphdb_builder.builder_utils.**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).

graphdb_builder.builder_utils.**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).

graphdb_builder.builder_utils.**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.

---

graphdb_builder.builder_utils.**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

graphdb_builder.builder_utils.**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.

---

graphdb_builder.builder_utils.**get_full_path_directories**()
Reads Builder YAML configuration file and returns the full path of all directories. :return: Dictionary.

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**download_from_ftp**(*ftp_url*, *user*, *password*, *to*, *file_name*)

graphdb_builder.builder_utils.**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)

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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'.

graphdb_builder.builder_utils.**remove_directory**(*directory*)

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

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

Returns List of file names.

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

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

Returns List of folder names.

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

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

Returns List of folder names.

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

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

graphdb_builder.builder_utils.**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]
```

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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.

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

Returns Two strings: date and time.

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

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

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

graphdb_builder.builder_utils.**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

graphdb_builder.builder_utils.**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.

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

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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.

graphdb_builder.builder_utils.**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

graphdb_builder.builder_utils.**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

graphdb_builder.mapping.**reset_mapping**(*entity*)
Checks if mapping.tsv file exists and removes it.

Parameters **entity** (*str*) – entity label as defined in databases_config.yml

graphdb_builder.mapping.**mark_complete_mapping**(*entity*)
Checks if mapping.tsv file exists and renames it to complete_mapping.tsv.

Parameters **entity** (*str*) – entity label as defined in databases_config.yml

graphdb_builder.mapping.**getMappingFromOntology**(*ontology*, *source=None*)
Converts .tsv file with complete list of ontology identifiers and aliases, to dictionary with aliases as keys and ontology identifiers as values.

Parameters

- **ontology** (*str*) – ontology label as defined in ontologies_config.yml.
- **source** (*str or None*) – name of the source database for selecting aliases.

Returns Dictionary of aliases (keys) and ontology identifiers (values).

graphdb_builder.mapping.**getMappingFromDatabase**(*id_list*, *node*, *attribute_from='id'*, *attribute_to='name'*)

graphdb_builder.mapping.**getMappingForEntity**(*entity*)
Converts .tsv file with complete list of entity identifiers and aliases, to dictionary with aliases as keys and entity identifiers as values.

Parameters **entity** (*str*) – entity label as defined in databases_config.yml.

Returns Dictionary of aliases (keys) and entity identifiers (value).

graphdb_builder.mapping.**getMultipleMappingForEntity**(*entity*)
Converts .tsv file with complete list of entity identifiers and aliases, to dictionary with aliases to other databases as keys and entity identifiers as values.

Parameters **entity** (*str*) – entity label as defined in databases_config.yml.

Returns Dictionary of aliases (keys) and set of unique entity identifiers (values).

graphdb_builder.mapping.**get_STRING_mapping_url**(*db='STRING'*)
Get the url for downloading the mapping file from either STRING or STITCH

Parameters **db** (*str*) – Which database to get the url from: STRING or STITCH

Returns url where to download the mapping file

graphdb_builder.mapping.**getSTRINGMapping**(*source='BLAST_UniProt_AC'*, *download=True*, *db='STRING'*)
Parses database (db) and extracts relationships between identifiers to order databases (source).

Parameters

- **url** (*str*) – link to download database raw file.

- **source** (*str*) – name of the source database for selecting aliases.
- **download** (*bool*) – wether to download the file or not.
- **db** (*str*) – name of the database to be parsed.

Returns Dictionary of database identifers (keys) and set of unique aliases to other databases (values).

graphdb_builder.mapping.**buildMappingFromOBO**(*oboFile*, *ontology*)

Parses and extracts ontology idnetifiers, names and synonyms from raw file, and writes all the information to a .tsv file. :param str oboFile: path to ontology raw file. :param str ontology: ontology database acronym as defined in ontologies_config.yml.

graphdb_builder.mapping.**map_experiment_files**(*project_id*, *datasetPath*, *mapping*)

graphdb_builder.mapping.**map_experimental_data**(*data*, *mapping*)

graphdb_builder.mapping.**get_mapping_analytical_samples**(*project_id*)

8.1.3 Report Manager

Report Dash Apps

Basic App

class report_manager.apps.basicApp.**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.

Data Upload App

Data Upload

report_manager.apps.dataUpload.**get_data_upload_queries**()
Reads the YAML file containing the queries relevant to parsing of clinical data and returns a Python object (dict[dict]).

Returns Nested dictionary.

report_manager.apps.dataUpload.**get_new_subject_identifier**(*driver*)
Queries the database for the last subject identifier and returns a new sequential identifier.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **projectId** (*str*) – external project identifier (from the graph database).

Returns Subject identifier.

Return type str

report_manager.apps.dataUpload.**get_new_biosample_identifier**(*driver*)
Queries the database for the last biological sample internal identifier and returns a new sequential identifier.

Parameters **driver** – py2neo driver, which provides the connection to the neo4j graph database.

Returns Biological sample identifier.

report_manager.apps.dataUpload.**get_new_analytical_sample_identifier**(*driver*)
Queries the database for the last analytical sample internal identifier and returns a new sequential identifier. :param driver: py2neo driver, which provides the connection to the neo4j graph database.

Returns Analytical sample identifier.

report_manager.apps.dataUpload.**get_subjects_enrolled_in_project**(*driver*, *projectId*)
Extracts the number of subjects included in a given project.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **projectId** (*str*) – external project identifier (from the graph database).

Returns Number of subjects.

Return type Numpy ndarray

report_manager.apps.dataUpload.**check_samples_in_project**(*driver*, *projectId*)

report_manager.apps.dataUpload.**check_external_ids_in_db**(*driver*, *projectId*)

report_manager.apps.dataUpload.**remove_samples_nodes_db**(*driver*, *projectId*)

report_manager.apps.dataUpload.**create_new_subjects**(*driver*, *data*, *projectId*)

Parameters

- **driver** – py2neo driver, which provides the connection to the neo4j graph database.
- **data** – pandas Dataframe with clinical data as columns and samples as rows.

- **projectId** (*string*) – project identifier.

Returns Pandas DataFrame where new biological sample internal identifiers have been added.

report_manager.apps.dataUpload.**create_new_biosamples**(*driver*, *data*)

Parameters

- **driver** – py2neo driver, which provides the connection to the neo4j graph database.
- **data** – pandas Dataframe with clinical data as columns and samples as rows.

Returns Pandas DataFrame where new biological sample internal identifiers have been added.

report_manager.apps.dataUpload.**create_new_ansamples**(*driver*, *data*)

Parameters

- **driver** – py2neo driver, which provides the connection to the neo4j graph database.
- **data** – pandas Dataframe with clinical data as columns and samples as rows.

Returns Pandas DataFrame where new analytical sample internal identifiers have been added.

report_manager.apps.dataUpload.**create_experiment_internal_identifiers**(*driver*, *projectId*, *data*, *directory*, *filename*)

report_manager.apps.dataUpload.**create_mapping_cols_clinical**(*driver*, *data*, *directory*, *filename*, *separator='|'*)

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **data** – pandas Dataframe with clinical data as columns and samples as rows.
- **separator** (*str*) – character used to separate multiple entries in an attribute.

Returns Pandas Dataframe with all clinical data and graph database internal identifiers.

report_manager.apps.dataUpload.**get_project_information**(*driver*, *project_id*)

HomePage Stats App

HomePage Stats

report_manager.apps.homepageStats.**size_converter**(*value*)

Converts a given value to the highest possible unit, maintaining two decimals.

Parameters or float value (*int*) –

Returns String with converted value and units.

report_manager.apps.homepageStats.**get_query**()

Reads the YAML file containing the queries relevant for graph database stats, parses the given stream and returns a Python object (dict[dict]).

Returns Nested dictionary.

report_manager.apps.homepageStats.**get_db_schema**()
Retrieves the database schema

Returns network with all the database nodes and how they are related

report_manager.apps.homepageStats.**get_db_stats_data**()
Retrieves all the stats data from the graph database and returns them as a dictionary.

Returns Dictionary of dataframes.

report_manager.apps.homepageStats.**plot_store_size_components**(*dfs*, *title*, *args*)
Plots the store size of different components of the graph database, as a Pie Chart.

Parameters

- **dfs** (*dict*) – dictionary of json objects.
- **title** (*str*) – title of the Dash div where plot is located.
- **args** (*dict*) – see below.

Arguments

- **valueCol** (str) – name of the column with the values to be plotted.
- **textCol** (str) – name of the column containing information for the hoverinfo parameter.
- **height** (str) – height of the plot.
- **width** (str) – width of the plot.

Returns New Dash div containing title and pie chart.

report_manager.apps.homepageStats.**plot_node_rel_per_label**(*dfs*, *title*, *args*, *focus='nodes'*)
Plots the number of nodes or relationships (depending on 'focus') per label, contained in the grapha database.

Parameters

- **dfs** (*dict*) – dictionary of json objects.
- **title** (*str*) – title of the Dash div where plot is located.

Paeam str focus plot number of nodes per label ('nodes') or the number of relationships per type ('relationships').

Returns New Dash div containing title and barplot.

report_manager.apps.homepageStats.**indicator**(*color*, *text*, *id_value*)
Builds a new Dash div styled as a container, with borders and background.

Parameters

- **color** (*str*) – background color of the container (RGB or Hex colors).
- **text** (*str*) – name to be plotted inside the container.
- **id_value** (*str*) – identifier of the container.

Returns Dash div containing title and an html.P element.

report_manager.apps.homepageStats.**quick_numbers_panel**()
Creates a panel of Dash containers where an overviem of the graph database numbers can be plotted.

Returns List of Dash components.

DB Imports Stats App

DB Imports Stats

report_manager.apps.imports.**get_stats_data**(*filename*, *n=3*)
Reads graph database stats file and filters for the last 'n' full and partial independent imports, returning a Pandas DataFrame.

Parameters

- **filename** (*str*) – path to stats file (including filename and '.hdf' extension).
- **n** (*int*) – number of independent imports to plot.

Returns Pandas Dataframe with different entities and relationships as rows and columns:

report_manager.apps.imports.**select_last_n_imports**(*stats_file*, *n=3*)
Selects which independent full and partial imports should be plotted based on n.

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **n** (*int*) – number of independent imports to select.

Returns List of import ids to be plotted according to selection criterion.

report_manager.apps.imports.**remove_legend_duplicates**(*figure*)
Removes duplicated legend items.

Parameters **figure** – plotly graph object figure.

report_manager.apps.imports.**get_databases_entities_relationships**(*stats_file*, *key='full'*, *options='databases'*)
Builds dictionary from stats file. Depending on 'options', keys and values can differ. If *options* is set to 'dates', keys are dates of the imports and values are databases imported at each date; if 'databases', keys are databases and values are entities and relationships created from each database; if 'entities', keys are databases and values are entities created from each database; if 'relationships', keys are databases and values are relationships created from each database.

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **key** (*str*) – use only full, partial or both kinds of imports ('full', 'partial', 'all').
- **options** (*str*) – name of the variables to be used as keys in the output dictionary ('dates', 'databases', 'entities' or 'relationships').

Returns Dictionary.

report_manager.apps.imports.**set_colors**(*dictionary*)
This function takes the values in a dictionary and attributes them an RGB color.

Parameters **dictionary** (*dict*) – dictionary with variables to be attributed a color, as values.

Returns Dictionary where 'dictionary' values are keys and random RGB colors are the values.

report_manager.apps.imports.**get_dropdown_menu**(*fig*, *options_dict*, *add_button=True*, *equal_traces=True*, *number_traces=2*)

Builds a list for the dropdown menu, based on a plotly figure traces and a dictionary with the options to be used in the dropdown.

Parameters

- **fig** – plotly graph object figure.
- **options_dict** – dictionary where keys are used as dropdown options and values data points.
- **add_button** (*bool*) – add option to display all dropdown options simultaneously.
- **equal_traces** (*bool*) – defines if all dropdown options have the same number of traces each. If True, define 'number_traces' as well. If False, number of traces will be the same as the number of values for each 'options_dict' key.
- **number_traces** (*int*) – number of traces created for each 'options_dict' key.

Returns List of nested structures. Each dictionary within *updatemenus[0]['buttons'][0]* corresponds to one dropdown menu options and contains information on which traces are visible, label and method.

report_manager.apps.imports.**get_totals_per_date**(*stats_file*, *key='full'*, *import_types=False*)

Summarizes stats file to a Pandas DataFrame with import dates and total number of imported entities and relationships.

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **key** (*str*) – use only full or partial imports ('full', 'partial').
- **import_types** (*bool*) – breakdown importing stats into entities or relationships related.

Returns Pandas DataFrame with independent import dates as rows and imported numbers as columns.

report_manager.apps.imports.**get_imports_per_database_date**(*stats_file*)

Summarizes stats file to a Pandas DataFrame with import dates, databases and total number of imported entities and relationships per database.

Parameters **stats_file** – pandas DataFrame with stats data.

Returns Pandas DataFrame with independent import dates and databases as rows and imported numbers as columns.

report_manager.apps.imports.**plot_total_number_imported**(*stats_file*, *plot_title*)

Creates plot with overview of imports numbers per date.

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **plot_title** (*str*) – title of the plot.

Returns Line plot figure within the .

report_manager.apps.imports.**plot_total_numbers_per_date**(*stats_file*, *plot_title*)

Plots number of entities and relationships imported per date, with scaled markers reflecting numbers rations.

Parameters

- **stats_file** – pandas DataFrame with stats data.

- **plot_title** (*str*) – title of the plot.

Returns Scatter plot figure within the , with scalled markers.

report_manager.apps.imports.**plot_databases_numbers_per_date**(*stats_file*, *plot_title*, *key='full'*, *dropdown=False*, *dropdown_options='dates'*)

Grouped horizontal barplot showing the number of entities and relationships imported from each biomedical database.

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **plot_title** (*str*) – title of the plot.
- **key** (*str*) – use only full or partial imports ('full', 'partial').
- **dropdown** (*bool*) – add dropdown menu to figure or not.
- **dropdown_options** (*str*) – name of the variables to be used as options in the dropdown menu ('dates', 'databases', 'entities' or 'relationships').

Returns Horizontal barplot figure within the .

report_manager.apps.imports.**plot_import_numbers_per_database**(*stats_file*, *plot_title*, *key='full'*, *subplot_titles=('', '')*, *colors=True*, *plots_1='entities'*, *plots_2='relationships'*, *dropdown=True*, *dropdown_options='databases'*)

Creates plotly multiplot figure with breakdown of imported numbers and size of the respective files, per database and import type (entities or relationships).

Parameters

- **stats_file** – pandas DataFrame with stats data.
- **plot_title** (*str*) – title of the plot.
- **key** (*str*) – use only full or partial imports ('full', 'partial').
- **subplot_titles** (*tuple*) – title of the subplots (tuple of strings, one for each subplot).
- **colors** (*bool*) – define standard colors for entities and for relationships.
- **plots_1** (*str*) – name of the variable plotted.
- **plots_2** (*str*) – name of the variable plotted.
- **dropdown** (*bool*) – add dropdown menu to figure or not.
- **dropdown_options** (*str*) – name of the variables to be used as options in the dropdown menu ('dates', 'databases', 'entities' or 'relationships').

Returns Multi-scatterplot figure within the .

Initial App

Login App

Project App

Project Creation App

Project Creation

report_manager.apps.projectCreation.**get_project_creation_queries**()
Reads the YAML file containing the queries relevant to user creation, parses the given stream and returns a Python object (dict[dict]).

Returns Nested dictionary.

report_manager.apps.projectCreation.**check_if_node_exists**(*driver*, *node*, *node_property*, *value*)
Queries the graph database and checks if a node with a specific property and property value already exists. :param driver: py2neo driver, which provides the connection to the neo4j graph database. :type driver: py2neo driver :param str node: node to be matched in the database. :param str node_property: property of the node. :param value: property value. :type value: str, int, float or bool :return: Pandas dataframe with user identifier if User with node_property and value already exists, if User does not exist, returns and empty dataframe.

report_manager.apps.projectCreation.**get_new_project_identifier**(*driver*, *projectId*)
Queries the database for the last project external identifier and returns a new sequential identifier.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **projectId** (*str*) – internal project identifier (CPxxxxxxxxxxx).

Returns Project external identifier.

Return type str

report_manager.apps.projectCreation.**get_subject_number_in_project**(*driver*, *projectId*)
Extracts the number of subjects included in a given project.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **projectId** (*str*) – external project identifier (from the graph database).

Returns Integer with the number of subjects.

report_manager.apps.projectCreation.**create_new_project**(*driver*, *projectId*, *data*, *separator='|'*)
Creates a new project in the graph database, following the steps:

1. Retrieves new project external identifier and creates project node and relationships in the graph database.
2. Creates subjects, timepoints and intervention nodes.
3. Saves all the entities and relationships to tab-delimited files.

4. Returns the number of projects created and the project external identifier.

Parameters

- **driver** (*py2neo driver*) – py2neo driver, which provides the connection to the neo4j graph database.
- **projectId** (*str*) – internal project identifier (CPxxxxxxxxxxx).
- **data** – pandas Dataframe with project as row and other attributes as columns.
- **separator** (*str*) – character used to separate multiple entries in a project attribute.

Returns Two strings: number of projects created and the project external identifier.

app.py

dataset.py

index.py

knowledge.py

project.py

report.py

class report_manager.report.**Report**(*identifier*, *plots={}*)

Bases: object

property identifier

property plots

get_plot(*plot*)

update_plots(*plot*)

list_plots()

print_report(*directory*, *plot_format='pdf'*)

save_report(*directory*)

read_report(*directory*)

visualize_report(*environment*)

visualize_plot(*environment*, *plot_type*)

download_report(*directory*)

user.py

class report_manager.user.**User**(*username*)
Bases: object

find()

register(*password*)

verify_password(*password*)

utils.py

report_manager.utils.**copy_file_to_destination**(*cfile*, *destination*)

report_manager.utils.**send_message_to_slack_webhook**(*message*, *message_to*, *username='albsantosdel'*)

report_manager.utils.**send_email**(*message*, *subject*, *message_from*, *message_to*)

report_manager.utils.**compress_directory**(*name*, *directory*, *compression_format='zip'*)

report_manager.utils.**get_markdown_date**(*extra_text*)

report_manager.utils.**convert_html_to_dash**(*el*, *style=None*)

report_manager.utils.**extract_style**(*el*)

report_manager.utils.**get_image**(*figure*, *width*, *height*)

report_manager.utils.**parse_html**(*html_snippet*)

report_manager.utils.**hex2rgb**(*color*)

report_manager.utils.**getNumberText**(*num*)

report_manager.utils.**get_rgb_colors**(*n*)

report_manager.utils.**get_hex_colors**(*n*)

report_manager.utils.**convert_html_to_pdf**(*source_html*, *output_filename*)

worker.py

8.1.4 Analytics Core

Analysis

Analytics

analytics_core.analytics.analytics.**unit_vector**(*vector*)
Returns the unit vector of the vector. :param tuple vector: vector :return tuple unit_vector: unit vector

analytics_core.analytics.analytics.**flatten**(*t*, *my_list=[]*)
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]
```

analytics_core.analytics.analytics.**angle_between**(*v1*, *v2*)

Returns the angle in radians between vectors 'v1' and 'v2'

Parameters

- **v1** (*tuple*) – vector 1
- **v2** (*tuple*) – vector 2

Return float angle angle between two vectors in radians

Example:: angle = angle_between((1, 0, 0), (0, 1, 0))

analytics_core.analytics.analytics.**transform_into_wide_format**(*data*, *index*, *columns*, *values*, *extra=[]*)

This function converts a Pandas DataFrame from long to wide format using pandas pivot_table() function.

Parameters

- **data** – long-format Pandas DataFrame
- **index** (*list*) – columns that will be converted into the index
- **columns** (*str*) – column name whose unique values will become the new column names
- **values** (*str*) – column to aggregate
- **extra** (*list*) – additional columns to be kept as columns

Returns Wide-format pandas DataFrame

Example:

```
result = transform_into_wide_format(df, index='index', columns='x', values='y',␣
↪extra='group')
```

analytics_core.analytics.analytics.**transform_into_long_format**(*data*, *drop_columns*, *group*, *columns=['name', 'y']*)

Converts a Pandas DataDrame from wide to long format using pd.melt() function.

Parameters

- **data** – wide-format Pandas DataFrame
- **drop_columns** (*list*) – columns to be deleted
- **group** (*str or list*) – column(s) to use as identifier variables
- **columns** (*list*) – names to use for the 1)variable column, and for the 2)value column

Returns Long-format Pandas DataFrame.

Example:

```
result = transform_into_long_format(df, drop_columns=['sample', 'subject'], group=
↪'group', columns=['name','y'])
```

analytics_core.analytics.analytics.**get_ranking_with_markers**(*data*, *drop_columns*, *group*, *columns*, *list_markers*, *annotation={}*)

This function creates a long-format dataframe with features and values to be plotted together with disease biomarker annotations.

Parameters

- **data** – wide-format Pandas DataFrame with samples as rows and features as columns
- **drop_columns** (*list*) – columns to be deleted
- **group** (*str*) – column to use as identifier variables
- **columns** (*list*) – names to use for the 1)variable column, and for the 2)value column
- **list_markers** (*list*) – list of features from data, known to be markers associated to disease.
- **annotation** (*dict*) – markers, from list_markers, and associated diseases.

Returns Long-format pandas DataFrame with group identifiers as rows and columns: 'name' (identifier), 'y' (LFQ intensity), 'symbol' and 'size'.

Example:

```
result = get_ranking_with_markers(data, drop_columns=['sample', 'subject'], group=
↪'group', columns=['name', 'y'], list_markers, annotation={})
```

analytics_core.analytics.analytics.**extract_number_missing**(*data*, *min_valid*, *drop_cols=['sample']*, *group='group'*)

Counts how many valid values exist in each column and filters column labels with more valid values than the minimum threshold defined.

Parameters

- **data** – pandas DataFrame with group as rows and protein identifier as column.
- **group** (*str*) – column label containing group identifiers. If None, number of valid values is counted across all samples, otherwise is counted per unique group identifier.
- **min_valid** (*int*) – minimum number of valid values to be filtered.
- **drop_columns** (*list*) – column labels to be dropped.

Returns List of column labels above the threshold.

Example:

```
result = extract_number_missing(data, min_valid=3, drop_cols=['sample'], group=
↪'group')
```

analytics_core.analytics.analytics.**extract_percentage_missing**(*data*, *missing_max*, *drop_cols=['sample']*, *group='group'*, *how='all'*)

Extracts ratio of missing/valid values in each column and filters column labels with lower ratio than the minimum threshold defined.

Parameters

- **data** – pandas dataframe with group as rows and protein identifier as column.
- **group** (*str*) – column label containing group identifiers. If None, ratio is calculated across all samples, otherwise is calculated per unique group identifier.
- **missing_max** (*float*) – maximum ratio of missing/valid values to be filtered.
- **how** (*str*) – define if labels with a higher percentage of missing values than the threshold in any group ('any') or in all groups ('all') should be filtered

Returns List of column labels below the threshold.

Example:: result = extract_percentage_missing(data, missing_max=0.3, drop_cols=['sample'], group='group')

analytics_core.analytics.analytics.**imputation_KNN**(*data, drop_cols=['group', 'sample', 'subject'], group='group', cutoff=0.6, alone=True*)

k-Nearest Neighbors imputation for pandas dataframes with missing data. For more information visit https://github.com/iskandr/fancyimpute/blob/master/fancyimpute/knn.py.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **group** (*str*) – column label containing group identifiers.
- **drop_cols** (*list*) – column labels to be dropped. Final dataframe should only have gene/protein/etc identifiers as columns.
- **cutoff** (*float*) – minimum ratio of missing/valid values required to impute in each column.
- **alone** (*boolean*) – if True removes all columns with any missing values.

Returns Pandas dataframe with samples as rows and protein identifiers as columns.

Example:

```
result = imputation_KNN(data, drop_cols=['group', 'sample', 'subject'], group=
↪'group', cutoff=0.6, alone=True)
```

analytics_core.analytics.analytics.**imputation_mixed_norm_KNN**(*data, index_cols=['group', 'sample', 'subject'], shift=1.8, nstd=0.3, group='group', cutoff=0.6*)

Missing values are replaced in two steps: 1) using k-Nearest Neighbors we impute protein columns with a higher ratio of missing/valid values than the defined cutoff, 2) the remaining missing values are replaced by random numbers that are drawn from a normal distribution.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **group** (*str*) – column label containing group identifiers.
- **index_cols** (*list*) – list of column labels to be set as dataframe index.

- **shift** (*float*) – specifies the amount by which the distribution used for the random numbers is shifted downwards. This is in units of the standard deviation of the valid data.
- **nstd** (*float*) – defines the width of the Gaussian distribution relative to the standard deviation of measured values. A value of 0.5 would mean that the width of the distribution used for drawing random numbers is half of the standard deviation of the data.
- **cutoff** (*float*) – minimum ratio of missing/valid values required to impute in each column.

Returns Pandas dataframe with samples as rows and protein identifiers as columns.

Example:

```
result = imputation_mixed_norm_KNN(data, index_cols=['group', 'sample', 'subject
↪'], shift = 1.8, nstd = 0.3, group='group', cutoff=0.6)
```

analytics_core.analytics.analytics.**imputation_normal_distribution**(*data, index_cols=['group', 'sample', 'subject'], shift=1.8, nstd=0.3*)

Missing values will be replaced by random numbers that are drawn from a normal distribution. The imputation is done for each sample (across all proteins) separately. For more information visit http://www.coxdocs.org/doku.php?id=perseus:user:activities:matrixprocessing:imputation:replacemissingfromgaussian.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **index_cols** (*list*) – list of column labels to be set as dataframe index.
- **shift** (*float*) – specifies the amount by which the distribution used for the random numbers is shifted downwards. This is in units of the standard deviation of the valid data.
- **nstd** (*float*) – defines the width of the Gaussian distribution relative to the standard deviation of measured values. A value of 0.5 would mean that the width of the distribution used for drawing random numbers is half of the standard deviation of the data.

Returns Pandas dataframe with samples as rows and protein identifiers as columns.

Example:

```
result = imputation_normal_distribution(data, index_cols=['group', 'sample',
↪'subject'], shift = 1.8, nstd = 0.3)
```

analytics_core.analytics.analytics.**normalize_data_per_group**(*data, group, method='median'*)

This function normalizes the data by group using the selected method

Parameters

- **data** – DataFrame with the data to be normalized (samples x features)
- **group_col** – Column containing the groups
- **method** (*string*) – normalization method to choose among: median_polish, median, quantile, linear

Returns Pandas dataframe.

Example:

```
result = normalize_data_per_group(data, group='group' method='median')
```

analytics_core.analytics.analytics.**normalize_data**(*data*, *method='median_polish'*)
This function normalizes the data using the selected method

Parameters

- **data** – DataFrame with the data to be normalized (samples x features)
- **method** (*string*) – normalization method to choose among: median_polish, median, quantile, linear

Returns Pandas dataframe.

Example:

```
result = normalize_data(data, method='median_polish')
```

analytics_core.analytics.analytics.**median_normalization**(*data*)
This function normalizes each sample by using its median.

Parameters **data** –

Returns Pandas dataframe.

Example:

```
result = median_normalization(data)
```

analytics_core.analytics.analytics.**zscore_normalization**(*data*)
This function normalizes each sample by using its mean and standard deviation (mean=0, std=1).

Parameters **data** –

Returns Pandas dataframe.

Example:: data = pd.DataFrame({'a': [2,5,4,3,3], 'b':[4,4,6,5,3], 'c':[4,14,8,8,9]}) result = zscore_normalization(data) result

a b c

0 -1.154701 0.577350 0.577350 1 -0.484182 -0.665750 1.149932 2 -1.000000 0.000000 1.000000 3 -0.927173 -0.132453 1.059626 4 -0.577350 -0.577350 1.154701

analytics_core.analytics.analytics.**median_polish_normalization**(*data*, *max_iter=250*)
This function iteratively normalizes each sample and each feature to its median until medians converge.

Parameters

- **data** –
- **max_iter** (*int*) – number of maximum iterations to prevent infinite loop.

Returns Pandas dataframe.

Example:

```
result = median_polish_normalization(data, max_iter = 10)
```

analytics_core.analytics.analytics.**quantile_normalization**(*data*)
Applies quantile normalization to each column in pandas dataframe.

Parameters **data** – pandas dataframe with features as columns and samples as rows.

Returns Pandas dataframe

Example:

```
result = quantile_normalization(data)
```

analytics_core.analytics.analytics.**linear_normalization**(*data*, *method='l1'*, *axis=0*)

This function scales input data to a unit norm. For more information visit https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html.

Parameters

- **data** – pandas dataframe with samples as rows and features as columns.
- **method** (*str*) – norm to use to normalize each non-zero sample or non-zero feature (depends on axis).
- **axis** (*int*) – axis used to normalize the data along. If 1, independently normalize each sample, otherwise (if 0) normalize each feature.

Returns Pandas dataframe

Example:

```
result = linear_normalization(data, method = "l1", axis = 0)
```

analytics_core.analytics.analytics.**remove_group**(*data*)

Removes column with label 'group'.

Parameters **data** – pandas dataframe with one column labelled 'group'

Returns Pandas dataframe

Example:

```
result = remove_group(data)
```

analytics_core.analytics.analytics.**calculate_coefficient_variation**(*values*)

Compute the coefficient of variation, the ratio of the biased standard deviation to the mean, in percentage. For more information visit https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.variation.html.

Parameters **values** (*ndarray*) – numpy array

Returns The calculated variation along rows.

Return type ndarray

Example:

```
result = calculate_coefficient_variation()
```

analytics_core.analytics.analytics.**get_coefficient_variation**(*data*, *drop_columns*, *group*, *columns=['name', 'y']*)

Extracts the coefficients of variation in each group.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').

- **drop_columns** (*list*) – column labels to be dropped from the dataframe
- **group** (*str*) – column label containing group identifiers.
- **columns** (*list*) – names to use for the variable column(s), and for the value column(s)

Returns Pandas dataframe with columns 'name' (protein identifier), 'x' (coefficient of variation), 'y' (mean) and 'group'.

Exmaple:

```
result = get_coefficient_variation(data, drop_columns=['sample', 'subject'],␣
↪group='group')
```

analytics_core.analytics.analytics.**transform_proteomics_edgelist**(*df, index_cols=['group', 'sample', 'subject'], drop_cols=['sample'], group='group', identifier='identifier', extra_identifier='name', value_col='LFQ_intensity'*)

Transforms a long format proteomics matrix into a wide format

Parameters

- **df** – long-format pandas dataframe with columns 'group', 'sample', 'subject', 'identifier' (protein), 'name' (gene) and 'LFQ_intensity'.
- **index_cols** (*list*) – column labels to be be kept as index identifiers.
- **drop_cols** (*list*) – column labels to be dropped from the dataframe.
- **group** (*str*) – column label containing group identifiers.
- **identifier** (*str*) – column label containing feature identifiers.
- **extra_identifier** (*str*) – column label containing additional protein identifiers (e.g. gene names).
- **value_col** (*str*) – column label containing expression values.

Returns Pandas dataframe with samples as rows and protein identifiers (UniprotID~GeneName) as columns (with additional columns 'group', 'sample' and 'subject').

Example: df = transform_proteomics_edgelist(original, index_cols=['group', 'sample', 'subject'], drop_cols=['sample'], group='group', identifier='identifier', value_col='LFQ_intensity')

analytics_core.analytics.analytics.**get_proteomics_measurements_ready**(*df*, *index_cols=['group', 'sample', 'subject']*, *drop_cols=['sample']*, *group='group'*, *identifier='identifier'*, *extra_identifier='name'*, *imputation=True*, *method='distribution'*, *missing_method='percentage'*, *missing_per_group=True*, *missing_max=0.3*, *min_valid=1*, *value_col='LFQ_intensity'*, *shift=1.8*, *nstd=0.3*, *knn_cutoff=0.6*, *normalize=False*, *normalization_method='median'*, *normalize_group=False*)

Processes proteomics data extracted from the database: 1) filter proteins with high number of missing values (> missing_max or min_valid), 2) impute missing values. For more information on imputation method visit http://www.coxdocs.org/doku.php?id=perseus:user:activities:matrixprocessing:filterrows:filtervalidvaluesrows.

Parameters

- **df** – long-format pandas dataframe with columns 'group', 'sample', 'subject', 'identifier' (protein), 'name' (gene) and 'LFQ_intensity'.
- **index_cols** (*list*) – column labels to be be kept as index identifiers.
- **drop_cols** (*list*) – column labels to be dropped from the dataframe.
- **group** (*str*) – column label containing group identifiers.
- **identifier** (*str*) – column label containing feature identifiers.
- **extra_identifier** (*str*) – column label containing additional protein identifiers (e.g. gene names).
- **imputation** (*bool*) – if True performs imputation of missing values.
- **method** (*str*) – method for missing values imputation ('KNN', 'distribuition', or 'mixed')

- **missing_method** (*str*) – defines which expression rows are counted to determine if a column has enough valid values to survive the filtering process.
- **missing_per_group** (*bool*) – if True filter proteins based on valid values per group; if False filter across all samples.
- **missing_max** (*float*) – maximum ratio of missing/valid values to be filtered.
- **min_valid** (*int*) – minimum number of valid values to be filtered.
- **value_col** (*str*) – column label containing expression values.
- **shift** (*float*) – when using distribution imputation, the down-shift
- **nstd** (*float*) – when using distribution imputation, the width of the distribution
- **knn_cutoff** (*float*) – when using KNN imputation, the minimum percentage of valid values for which to use KNN imputation (i.e. 0.6 -> if 60% valid values use KNN, otherwise MinProb)

Returns Pandas dataframe with samples as rows and protein identifiers (UniprotID~GeneName) as columns (with additional columns 'group', 'sample' and 'subject').

Example 1:

```
result = get_proteomics_measurements_ready(df, index_cols=['group', 'sample',
↪'subject'], drop_cols=['sample'], group='group', identifier='identifier', extra_
↪identifier='name', imputation=True, method = 'distribution', missing_method =
↪'percentage', missing_per_group=True, missing_max = 0.3, value_col='LFQ_
↪intensity')
```

Example 2:

```
result = get_proteomics_measurements_ready(df, index_cols=['group', 'sample',
↪'subject'], drop_cols=['sample'], group='group', identifier='identifier', extra_
↪identifier='name', imputation = True, method = 'mixed', missing_method = 'at_
↪least_x', missing_per_group=False, min_valid=5, value_col='LFQ_intensity')
```

analytics_core.analytics.analytics.**get_clinical_measurements_ready**(*df*, *subject_id='subject'*, *sample_id='biological_sample'*, *group_id='group'*, *columns=['clinical_variable']*, *values='values'*, *extra=['group']*, *imputation=True*, *imputation_method='KNN'*)

Processes clinical data extracted from the database by converting dataframe to wide-format and imputing missing values.

Parameters

- **df** – long-format pandas dataframe with columns 'group', 'biological_sample', 'subject', 'clinical_variable', 'value'.
- **subject_id** (*str*) – column label containing subject identifiers.

- **sample_id** (*str*) – column label containing biological sample identifiers.
- **group_id** (*str*) – column label containing group identifiers.
- **columns** (*list*) – column name whose unique values will become the new column names
- **values** (*str*) – column label containing clinical variable values.
- **extra** (*list*) – additional column labels to be kept as columns
- **imputation** (*bool*) – if True performs imputation of missing values.
- **imputation_method** (*str*) – method for missing values imputation ('KNN', 'distribuition', or 'mixed').

Returns Pandas dataframe with samples as rows and clinical variables as columns (with additional columns 'group', 'subject' and 'biological_sample').

Example:

```
result = get_clinical_measurements_ready(df, subject_id='subject', sample_id=
↪'biological_sample', group_id='group', columns=['clinical_variable'], values=
↪'values', extra=['group'], imputation=True, imputation_method='KNN')
```

analytics_core.analytics.analytics.**get_summary_data_matrix**(*data*)

Returns some statistics on the data matrix provided.

Parameters **data** – pandas dataframe.

Returns dictionary with the type of statistics as key and the statistic as value in the shape of a pandas data frame

Example:

```
result = get_summary_data_matrix(data)
```

analytics_core.analytics.analytics.**check_equal_variances**(*data, drop_cols=['group', 'sample', 'subject'], group_col='group', alpha=0.05*)

analytics_core.analytics.analytics.**check_normality**(*data, drop_cols=['group', 'sample', 'subject'], group_col='group', alpha=0.05*)

analytics_core.analytics.analytics.**run_pca**(*data, drop_cols=['sample', 'subject'], group='group', components=2, dropna=True*)

Performs principal component analysis and returns the values of each component for each sample and each protein, and the loadings for each protein. For information visit https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **drop_cols** (*list*) – column labels to be dropped from the dataframe.
- **group** (*str*) – column label containing group identifiers.
- **components** (*int*) – number of components to keep.
- **dropna** (*bool*) – if True removes all columns with any missing values.

Returns Two dictionaries: 1) two pandas dataframes (first one with components values, the second with the components vectors for each protein), 2) xaxis and yaxis titles with components loadings for plotly.

Example:

```
result = run_pca(data, drop_cols=['sample', 'subject'], group='group',␣
↪components=2, dropna=True)
```

analytics_core.analytics.analytics.**run_tsne**(*data, drop_cols=['sample', 'subject'], group='group', components=2, perplexity=40, n_iter=1000, init='pca', dropna=True*)

Performs t-distributed Stochastic Neighbor Embedding analysis. For more information visit https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **drop_cols** (*list*) – column labels to be dropped from the dataframe.
- **group** (*str*) – column label containing group identifiers.
- **components** (*int*) – dimension of the embedded space.
- **perplexity** (*int*) – related to the number of nearest neighbors that is used in other manifold learning algorithms. Consider selecting a value between 5 and 50.
- **n_iter** (*int*) – maximum number of iterations for the optimization (at least 250).
- **init** (*str*) – initialization of embedding ('random', 'pca' or numpy array of shape n_samples x n_components).
- **dropna** (*bool*) – if True removes all columns with any missing values.

Returns Two dictionaries: 1) pandas dataframe with embedding vectors, 2) xaxis and yaxis titles for plotly.

Example:

```
result = run_tsne(data, drop_cols=['sample', 'subject'], group='group',␣
↪components=2, perplexity=40, n_iter=1000, init='pca', dropna=True)
```

analytics_core.analytics.analytics.**run_umap**(*data, drop_cols=['sample', 'subject'], group='group', n_neighbors=10, min_dist=0.3, metric='cosine', dropna=True*)

Performs Uniform Manifold Approximation and Projection. For more information vist https://umap-learn.readthedocs.io.

Parameters

- **data** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **drop_cols** (*list*) – column labels to be dropped from the dataframe.
- **group** (*str*) – column label containing group identifiers.
- **n_neighbors** (*int*) – number of neighboring points used in local approximations of manifold structure.

- **min_dist** (*float*) – controls how tightly the embedding is allowed compress points together.
- **metric** (*str*) – metric used to measure distance in the input space.
- **dropna** (*bool*) – if True removes all columns with any missing values.

Returns Two dictionaries: 1) pandas dataframe with embedding of the training data in low-dimensional space, 2) xaxis and yaxis titles for plotly.

Example:

```
result = run_umap(data, drop_cols=['sample', 'subject'], group='group', n_
↪neighbors=10, min_dist=0.3, metric='cosine', dropna=True)
```

analytics_core.analytics.analytics.**calculate_correlations**(*x*, *y*, *method='pearson'*)

Calculates a Spearman (nonparametric) or a Pearson (parametric) correlation coefficient and p-value to test for non-correlation.

Parameters

- **x** (*ndarray*) – array 1
- **y** (*ndarray*) – array 2
- **method** (*str*) – chooses which kind of correlation method to run

Returns Tuple with two floats, correlation coefficient and two-tailed p-value.

Example:

```
result = calculate_correlations(x, y, method='pearson')
```

analytics_core.analytics.analytics.**apply_pvalue_correction**(*pvalues*, *alpha=0.05*, *method='bonferroni'*)

Performs p-value correction using the specified method. For more information visit https://www.statsmodels.org/dev/generated/statsmodels.stats.multitest.multipletests.html.

Parameters

- **pvalues** (*ndarray*) – et of p-values of the individual tests.
- **alpha** (*float*) – error rate.
- **method** (*str*) – method of p-value correction: - bonferroni : one-step correction - sidak : one-step correction - holm-sidak : step down method using Sidak adjustments - holm : step-down method using Bonferroni adjustments - simes-hochberg : step-up method (independent) - hommel : closed method based on Simes tests (non-negative) - fdr_bh : Benjamini/Hochberg (non-negative) - fdr_by : Benjamini/Yekutieli (negative) - fdr_tsbh : two stage fdr correction (non-negative) - fdr_tsbky : two stage fdr correction (non-negative)

Returns Tuple with two arrays, boolen for rejecting H0 hypothesis and float for adjusted p-value.

Exmaple:

```
result = apply_pvalue_correction(pvalues, alpha=0.05, method='bonferroni')
```

analytics_core.analytics.analytics.**apply_pvalue_fdrcorrection**(*pvalues*, *alpha=0.05*, *method='indep'*)

Performs p-value correction for false discovery rate. For more information visit https://www.statsmodels.org/devel/generated/statsmodels.stats.multitest.fdrcorrection.html.

Parameters

- **pvalues** (*ndarray*) – et of p-values of the individual tests.
- **alpha** (*float*) – error rate.
- **method** (*str*) – method of p-value correction ('indep', 'negcorr').

Returns Tuple with two arrays, boolen for rejecting H0 hypothesis and float for adjusted p-value.

Exmaple:

```
result = apply_pvalue_fdrcorrection(pvalues, alpha=0.05, method='indep')
```

analytics_core.analytics.analytics.**apply_pvalue_twostage_fdrcorrection**(*pvalues*, *alpha=0.05*, *method='bh'*)

Iterated two stage linear step-up procedure with estimation of number of true hypotheses. For more information visit https://www.statsmodels.org/dev/generated/statsmodels.stats.multitest.fdrcorrection_twostage.html.

Parameters

- **pvalues** (*ndarray*) – et of p-values of the individual tests.
- **alpha** (*float*) – error rate.
- **method** (*str*) – method of p-value correction ('bky', 'bh').

Returns Tuple with two arrays, boolen for rejecting H0 hypothesis and float for adjusted p-value.

Exmaple:

```
result = apply_pvalue_twostage_fdrcorrection(pvalues, alpha=0.05, method='bh')
```

analytics_core.analytics.analytics.**apply_pvalue_permutation_fdrcorrection**(*df*, *observed_pvalues*, *group*, *alpha=0.05*, *permutations=50*)

This function applies multiple hypothesis testing correction using a permutation-based false discovery rate approach.

Parameters

- **df** – pandas dataframe with samples as rows and features as columns.
- **oberved_pvalues** – pandas Series with p-values calculated on the originally measured data.
- **group** (*str*) – name of the column containing group identifiers.
- **alpha** (*float*) – error rate. Values velow alpha are considered significant.
- **permutations** (*int*) – number of permutations to be applied.

Returns Pandas dataframe with adjusted p-values and rejected columns.

Example:

```
result = apply_pvalue_permutation_fdrcorrection(df, observed_pvalues, group='group
↪', alpha=0.05, permutations=50)
```

analytics_core.analytics.analytics.**get_counts_permutation_fdr**(*value*, *random*, *observed*, *n*, *alpha*)

Calculates local FDR values (q-values) by computing the fraction of accepted hits from the permuted data over accepted hits from the measured data normalized by the total number of permutations.

Parameters

- **value** (*float*) – computed p-value on measured data for a feature.
- **random** (*ndarray*) – p-values computed on the permuted data.
- **observed** – pandas Series with p-values calculated on the originally measured data.
- **n** (*int*) – number of permutations to be applied.
- **alpha** (*float*) – error rate. Values velow alpha are considered significant.

Returns Tuple with q-value and boolean for H0 rejected.

Example:

```
result = get_counts_permutation_fdr(value, random, observed, n=250, alpha=0.05)
```

analytics_core.analytics.analytics.**convertToEdgeList**(*data*, *cols*)

This function converts a pandas dataframe to an edge list where index becomes the source nodes and columns the target nodes.

Parameters

- **data** – pandas dataframe.
- **cols** (*list*) – names for dataframe columns.

Returns Pandas dataframe with columns cols.

analytics_core.analytics.analytics.**run_correlation**(*df*, *alpha=0.05*, *subject='subject'*, *group='group'*, *method='pearson'*, *correction='fdr_bh'*)

This function calculates pairwise correlations for columns in dataframe, and returns it in the shape of a edge list with 'weight' as correlation score, and the ajusted p-values.

Parameters

- **df** – pandas dataframe with samples as rows and features as columns.
- **subject** (*str*) – name of column containing subject identifiers.
- **group** (*str*) – name of column containing group identifiers.
- **method** (*str*) – method to use for correlation calculation ('pearson', 'spearman').
- **alpha** (*floar*) – error rate. Values velow alpha are considered significant.
- **correction** (*string*) – type of correction see apply_pvalue_correction for methods

Returns Pandas dataframe with columns: 'node1', 'node2', 'weight', 'padj' and 'rejected'.

Example:

```
result = run_correlation(df, alpha=0.05, subject='subject', group='group', method=
↪'pearson', correction='fdr_bh')
```

analytics_core.analytics.analytics.**run_multi_correlation**(*df_dict, alpha=0.05, subject='subject', on=['subject', 'biological_sample'], group='group', method='pearson', correction='fdr_bh'*)

This function merges all input dataframes and calculates pairwise correlations for all columns.

Parameters

- **df_dict** (*dict*) – dictionary of pandas dataframes with samples as rows and features as columns.
- **subject** (*str*) – name of the column containing subject identifiers.
- **group** (*str*) – name of the column containing group identifiers.
- **on** (*list*) – column names to join dataframes on (must be found in all dataframes).
- **method** (*str*) – method to use for correlation calculation ('pearson', 'spearman').
- **alpha** (*float*) – error rate. Values velow alpha are considered significant.
- **correction** (*string*) – type of correction see apply_pvalue_correction for methods

Returns Pandas dataframe with columns: 'node1', 'node2', 'weight', 'padj' and 'rejected'.

Example:

```
result = run_multi_correlation(df_dict, alpha=0.05, subject='subject', on=[
↪'subject', 'biological_sample'] , group='group', method='pearson', correction=
↪'fdr_bh')
```

analytics_core.analytics.analytics.**calculate_rm_correlation**(*df, x, y, subject*)

Computes correlation and p-values between two columns a and b in df.

Parameters

- **df** – pandas dataframe with subjects as rows and two features and columns.
- **x** (*str*) – feature a name.
- **y** (*str*) – feature b name.
- **subject** – column name containing the covariate variable.

Returns Tuple with values for: feature a, feature b, correlation, p-value and degrees of freedom.

Example:

```
result = calculate_rm_correlation(df, x='feature a', y='feature b', subject=
↪'subject')
```

analytics_core.analytics.analytics.**run_rm_correlation**(*df, alpha=0.05, subject='subject', correction='fdr_bh'*)

Computes pairwise repeated measurements correlations for all columns in dataframe, and returns results as an edge list with 'weight' as correlation score, p-values, degrees of freedom and ajusted p-values.

Parameters

- **df** – pandas dataframe with samples as rows and features as columns.
- **subject** (*str*) – name of column containing subject identifiers.

- **alpha** (*float*) – error rate. Values velow alpha are considered significant.
- **correction** (*string*) – type of correction type see apply_pvalue_correction for methods

Returns Pandas dataframe with columns: 'node1', 'node2', 'weight', 'pvalue', 'dof', 'padj' and 'rejected'.

Example:

```
result = run_rm_correlation(df, alpha=0.05, subject='subject', correction='fdr_bh
↪')
```

analytics_core.analytics.analytics.**run_efficient_correlation**(*data*, *method='pearson'*)

Calculates pairwise correlations and returns lower triangle of the matrix with correlation values and p-values.

Parameters

- **data** – pandas dataframe with samples as index and features as columns (numeric data only).
- **method** (*str*) – method to use for correlation calculation ('pearson', 'spearman').

Returns Two numpy arrays: correlation and p-values.

Example:

```
result = run_efficient_correlation(data, method='pearson')
```

analytics_core.analytics.analytics.**calculate_ttest_samr**(*df*, *labels*, *n=2*, *s0=0*, *paired=False*)

Calculates modified T-test using 'samr' R package.

Parameters

- **df** – pandas dataframe with group as columns and protein identifier as rows
- **abels** (*list*) – integers reflecting the group each sample belongs to (e.g. group1 = 1, group2 = 2)
- **n** (*int*) – number of samples
- **s0** (*float*) – exchangeability factor for denominator of test statistic
- **paired** (*bool*) – True if samples are paired

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(group1)', 'log2FC', 'FC', 't-statistics', 'p-value'.

Example:

```
result = calculate_ttest_samr(df, labels, n=2, s0=0.1, paired=False)
```

analytics_core.analytics.analytics.**calculate_ttest**(*df*, *condition1*, *condition2*, *paired=False*, *is_logged=True*, *non_par=False*, *tail='two-sided'*, *correction='auto'*, *r=0.707*)

Calculates the t-test for the means of independent samples belonging to two different groups. For more information visit https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html.

Parameters

- **df** – pandas dataframe with groups and subjects as rows and protein identifier as column.

- **condition1** (*str*) – identifier of first group.
- **condition2** (*str*) – ientifier of second group.
- **is_logged** (*bool*) – data is logged transformed
- **non_par** (*bool*) – if True, normality and variance equality assumptions are checked and non-parametric test Mann Whitney U test if not passed

Returns Tuple with t-statistics, two-tailed p-value, mean of first group, mean of second group and logfc.

Example:

```
result = calculate_ttest(df, 'group1', 'group2')
```

analytics_core.analytics.analytics.**calculate_THSD**(*df*, *column*, *group='group'*, *alpha=0.05*, *is_logged=True*)

Pairwise Tukey-HSD posthoc test using pingouin stats. For more information visit https://pingouin-stats.org/generated/pingouin.pairwise_tukey.html

Parameters

- **df** – pandas dataframe with group and protein identifier as columns
- **column** (*str*) – column containing the protein identifier
- **group** (*str*) – column label containing the between factor
- **alpha** (*float*) – significance level

Returns Pandas dataframe.

Example:

```
result = calculate_THSD(df, column='HBG2~P69892', group='group', alpha=0.05)
```

analytics_core.analytics.analytics.**calculate_pairwise_ttest**(*df*, *column*, *subject='subject'*, *group='group'*, *correction='none'*, *is_logged=True*)

Performs pairwise t-test using pingouin, as a posthoc test, and calculates fold-changes. For more information visit https://pingouin-stats.org/generated/pingouin.pairwise_ttests.html.

Parameters

- **df** – pandas dataframe with subject and group as rows and protein identifier as column.
- **column** (*str*) – column label containing the dependant variable
- **subject** (*str*) – column label containing subject identifiers
- **group** (*str*) – column label containing the between factor
- **correction** (*str*) – method used for testing and adjustment of p-values.

Returns Pandas dataframe with means, standard deviations, test-statistics, degrees of freedom and effect size columns.

Example:

```
result = calculate_pairwise_ttest(df, 'protein a', subject='subject', group='group
↪', correction='none')
```

analytics_core.analytics.analytics.**complement_posthoc**(*posthoc*, *identifier*, *is_logged*)

Calculates fold-changes after posthoc test.

Parameters

- **posthoc** – pandas dataframe from posthoc test. Should have at least columns 'mean(group1)' and 'mean(group2)'.
- **identifier** (*str*) – feature identifier.

Returns Pandas dataframe with additional columns 'identifier', 'log2FC' and 'FC'.

analytics_core.analytics.analytics.**calculate_dabest**(*df*, *idx*, *x*, *y*, *paired=False*, *id_col=None*, *test='mean_diff'*)

Parameters

- **df** –
- **idx** –
- **x** –
- **y** –
- **paired** –
- **id_col** –
- **test** –

Returns

analytics_core.analytics.analytics.**calculate_anova_samr**(*df*, *labels*, *s0=0*)

Calculates modified one-way ANOVA using 'samr' R package.

Parameters

- **df** – pandas dataframe with group as columns and protein identifier as rows
- **labels** (*list*) – integers reflecting the group each sample belongs to (e.g. group1 = 1, group2 = 2, group3 = 3)
- **s0** (*float*) – exchangeability factor for denominator of test statistic

Returns Pandas dataframe with protein identifiers and F-statistics.

Example:

```
result = calculate_anova_samr(df, labels, s0=0.1)
```

analytics_core.analytics.analytics.**calculate_anova**(*df*, *column*, *group='group'*)

Calculates one-way ANOVA using pingouin.

Parameters

- **df** – pandas dataframe with group as rows and protein identifier as column
- **column** (*str*) – name of the column in df to run ANOVA on
- **group** (*str*) – column with group identifiers

Returns Tuple with t-statistics and p-value.

analytics_core.analytics.analytics.**calculate_repeated_measures_anova**(*df*, *column*, *subject='subject'*, *group='group'*)

One-way and two-way repeated measures ANOVA using pingouin stats.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifier as column. Data must be in long-format for two-way repeated measures.
- **column** (*str*) – column label containing the dependant variable
- **subject** (*str*) – column label containing subject identifiers
- **group** (*str*) – column label containing the within factor

Returns Tuple with protein identifier, t-statistics and p-value.

Example:

```
result = calculate_repeated_measures_anova(df, 'protein a', subject='subject',␣
↪group='group')
```

analytics_core.analytics.analytics.**get_max_permutations**(*df*, *group='group'*)

Get maximum number of permutations according to number of samples.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns
- **group** (*str*) – column with group identifiers

Returns Maximum number of permutations.

Return type int

analytics_core.analytics.analytics.**check_is_paired**(*df*, *subject*, *group*)

Check if samples are paired.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **subject** (*str*) – column with subject identifiers
- **group** (*str*) – column with group identifiers

Returns True if paired samples.

Return type bool

analytics_core.analytics.analytics.**run_dabest**(*df*, *drop_cols=['sample']*, *subject='subject'*, *group='group'*, *test='mean_diff'*)

Parameters

- **df** –
- **drop_cols** (*list*) –
- **subject** (*str*) –
- **group** (*str*) –

- **test** (*str*) –

Returns Pandas dataframe

analytics_core.analytics.analytics.**run_anova**(*df, alpha=0.05, drop_cols=['sample', 'subject'], subject='subject', group='group', permutations=0, correction='fdr_bh', is_logged=True, non_par=False*)

Performs statistical test for each protein in a dataset. Checks what type of data is the input (paired, unpaired or repeated measurements) and performs posthoc tests for multiclass data. Multiple hypothesis correction uses permutation-based if permutations>0 and Benjamini/Hochberg if permutations=0.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **subject** (*str*) – column with subject identifiers
- **group** (*str*) – column with group identifiers
- **drop_cols** (*list*) – column labels to be dropped from the dataframe
- **alpha** (*float*) – error rate for multiple hypothesis correction
- **permutations** (*int*) – number of permutations used to estimate false discovery rates.
- **non_par** (*bool*) – if True, normality and variance equality assumptions are checked and non-parametric test Mann Whitney U test if not passed

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(group2)', 'Log2FC', 'std_error', 'tail', 't-statistics', 'posthoc pvalue', 'effsize', 'efftype', 'FC', 'rejected', 'F-statistics', 'p-value', 'correction', '-log10 p-value', and 'method'.

Example:

```
result = run_anova(df, alpha=0.05, drop_cols=["sample",'subject'], subject=
↪'subject', group='group', permutations=50)
```

analytics_core.analytics.analytics.**correct_pairwise_ttest**(*df, alpha, correction='fdr_bh'*)

analytics_core.analytics.analytics.**run_repeated_measurements_anova**(*df, alpha=0.05, drop_cols=['sample'], subject='subject', group='group', permutations=50, correction='fdr_bh', is_logged=True*)

Performs repeated measurements anova and pairwise posthoc tests for each protein in dataframe.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **subject** (*str*) – column with subject identifiers
- **group** (*srt*) – column with group identifiers

- **drop_cols** (*list*) – column labels to be dropped from the dataframe
- **alpha** (*float*) – error rate for multiple hypothesis correction
- **permutations** (*int*) – number of permutations used to estimate false discovery rates

Returns Pandas dataframe

Example:

```
result = run_repeated_measurements_anova(df, alpha=0.05, drop_cols=['sample'],␣
↪subject='subject', group='group', permutations=50)
```

analytics_core.analytics.analytics.**format_anova_table**(*df*, *aov_results*, *pairwise_results*, *pairwise_cols*, *group*, *permutations*, *alpha*, *correction*)

Performs p-value correction (permutation-based and FDR) and converts pandas dataframe into final format.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **aov_results** (*list[tuple]*) – list of tuples with anova results (one tuple per feature).
- **pairwise_results** (*list[dataframes]*) – list of pandas dataframes with posthoc tests results
- **group** (*str*) – column with group identifiers
- **alpha** (*float*) – error rate for multiple hypothesis correction
- **permutations** (*int*) – number of permutations used to estimate false discovery rates

Returns Pandas dataframe

analytics_core.analytics.analytics.**run_ttest**(*df*, *condition1*, *condition2*, *alpha=0.05*, *drop_cols=['sample']*, *subject='subject'*, *group='group'*, *paired=False*, *correction='fdr_bh'*, *permutations=50*, *is_logged=True*, *non_par=False*)

Runs t-test (paired/unpaired) for each protein in dataset and performs permutation-based (if permutations>0) or Benjamini/Hochberg (if permutations=0) multiple hypothesis correction.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **condition1** (*str*) – first of two conditions of the independent variable
- **condition2** (*str*) – second of two conditions of the independent variable
- **subject** (*str*) – column with subject identifiers
- **group** (*str*) – column with group identifiers (independent variable)
- **drop_cols** (*list*) – column labels to be dropped from the dataframe
- **paired** (*bool*) – paired or unpaired samples
- **correction** (*str*) – method of pvalue correction see apply_pvalue_correction for methods
- **alpha** (*float*) – error rate for multiple hypothesis correction

- **permutations** (*int*) – number of permutations used to estimate false discovery rates.
- **is_logged** (*bool*) – data is log-transformed
- **non_par** (*bool*) – if True, normality and variance equality assumptions are checked and non-parametric test Mann Whitney U test if not passed

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(group2)', 'std(group1)', 'std(group2)', 'Log2FC', 'FC', 'rejected', 'T-statistics', 'p-value', 'correction', '-log10 p-value', and 'method'.

Example:

```
result = run_ttest(df, condition1='group1', condition2='group2', alpha = 0.05,␣
↪drop_cols=['sample'], subject='subject', group='group', paired=False,␣
↪correction='fdr_bh', permutations=50)
```

analytics_core.analytics.analytics.**define_samr_method**(*df*, *subject*, *group*, *drop_cols*)

Method to identify the correct problem type to run with SAMR

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **subject** (*str*) – column with subject identifiers
- **group** (*str*) – column with group identifiers
- **droop_cols** (*str*) – columns to be dropped

Returns tuple with the method to be used (One Class, Two class paired, Two class unpaired or Multiclass) and the labels (conditions)

Example:

```
method, labels = define_samr_method(df, subject, group)
```

analytics_core.analytics.analytics.**calculate_pvalue_from_tstats**(*tstat*, *dfn*, *dfk*)

Calculate two-tailed p-values from T- or F-statistics.

tstat: T/F distribution dfn: degrees of freedrom *n* (values) per protein (keys), i.e. number of obervations - number of groups (dict) dfk: degrees of freedrom *n* (values) per protein (keys), i.e. number of groups - 1 (dict)

analytics_core.analytics.analytics.**run_samr**(*df*, *subject='subject'*, *group='group'*, *drop_cols=['subject'*, *'sample']*, *alpha=0.05*, *s0='null'*, *permutations=250*, *fc=0*, *is_logged=True*, *localfdr=False*)

Python adaptation of the 'samr' R package for statistical tests with permutation-based correction and s0 parameter. For more information visit https://cran.r-project.org/web/packages/samr/samr.pdf. The method only runs if R is installed and permutations is higher than 0, otherwise ANOVA.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns (with additional columns 'group', 'sample' and 'subject').
- **subject** (*str*) – column with subject identifiers
- **group** (*str*) – column with group identifiers
- **drop_cols** (*list*) – columnlabels to be dropped from the dataframe
- **alpha** (*float*) – error rate for multiple hypothesis correction

- **s0** (*float*) – exchangeability factor for denominator of test statistic
- **permutations** (*int*) – number of permutations used to estimate false discovery rates. If number of permutations is equal to zero, the function will run anova with FDR Benjamini/Hochberg correction.
- **fc** (*float*) – minimum fold change to define practical significance (needed when computing delta table)

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(group2)', 'Log2FC', 'FC', 'T-statistics', 'p-value', 'padj', 'correction', '-log10 p-value', 'rejected' and 'method'

Example:

```
result = run_samr(df, subject='subject', group='group', drop_cols=['subject',
↪'sample'], alpha=0.05, s0=1, permutations=250, fc=0)
```

analytics_core.analytics.analytics.**calculate_discriminant_lines**(*result*)

analytics_core.analytics.analytics.**run_fisher**(*group1*, *group2*, *alternative='two-sided'*)

annotated not-annotated group1 a b group2 c d ——————————

group1 = [a, b] group2 = [c, d]

odds, pvalue = stats.fisher_exact([[a, b], [c, d]])

analytics_core.analytics.analytics.**run_kolmogorov_smirnov**(*dist1*, *dist2*, *alternative='two-sided'*)

Compute the Kolmogorov-Smirnov statistic on 2 samples. See https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html

Parameters

- **dist1** (*list*) – sequence of 1-D ndarray (first distribution to compare) drawn from a continuous distribution
- **dist2** (*list*) – sequence of 1-D ndarray (second distribution to compare) drawn from a continuous distribution
- **alternative** (*str*) – defines the alternative hypothesis (default is 'two-sided'): * **'two-sided'** * **'less'** * **'greater'**

Returns statistic float and KS statistic pvalue float Two-tailed p-value.

Example:

```
result = run_kolmogorov_smirnov(dist1, dist2, alternative='two-sided')
```

analytics_core.analytics.analytics.**run_site_regulation_enrichment**(*regulation_data, annotation, identifier='identifier', groups=['group1', 'group2'], annotation_col='annotation', reject_col='rejected', group_col='group', method='fisher', regex='(\\w+~.+)_\\w\\d+\\-\\w+', correction='fdr_bh'*)

This function runs a simple enrichment analysis for significantly regulated protein sites in a dataset.

Parameters

- **regulation_data** – pandas dataframe resulting from differential regulation analysis.
- **annotation** – pandas dataframe with annotations for features (columns: 'annotation', 'identifier' (feature identifiers), and 'source').
- **identifier** (*str*) – name of the column from annotation containing feature identifiers.
- **groups** (*list*) – column names from regulation_data containing group identifiers.
- **annotation_col** (*str*) – name of the column from annotation containing annotation terms.
- **reject_col** (*str*) – name of the column from regulatio_data containing boolean for rejected null hypothesis.
- **group_col** (*str*) – column name for new column in annotation dataframe determining if feature belongs to foreground or background.
- **method** (*str*) – method used to compute enrichment (only 'fisher' is supported currently).
- **regex** (*str*) – how to extract the annotated identifier from the site identifier

Returns Pandas dataframe with columns: 'terms', 'identifiers', 'foreground', 'background', 'pvalue', 'padj' and 'rejected'.

Example:

```
result = run_site_regulation_enrichment(regulation_data, annotation, identifier=
↪'identifier', groups=['group1', 'group2'], annotation_col='annotation', reject_
↪col='rejected', group_col='group', method='fisher', match="(\w+~.+)_\w\d+\-\w+")
```

analytics_core.analytics.analytics.**run_regulation_enrichment**(*regulation_data, annotation, identifier='identifier', groups=['group1', 'group2'], annotation_col='annotation', reject_col='rejected', group_col='group', method='fisher', correction='fdr_bh'*)

This function runs a simple enrichment analysis for significantly regulated features in a dataset.

Parameters

- **regulation_data** – pandas dataframe resulting from differential regulation analysis.
- **annotation** – pandas dataframe with annotations for features (columns: 'annotation', 'identifier' (feature identifiers), and 'source').
- **identifier** (*str*) – name of the column from annotation containing feature identifiers.
- **groups** (*list*) – column names from regulation_data containing group identifiers.
- **annotation_col** (*str*) – name of the column from annotation containing annotation terms.
- **reject_col** (*str*) – name of the column from regulatio_data containing boolean for rejected null hypothesis.
- **group_col** (*str*) – column name for new column in annotation dataframe determining if feature belongs to foreground or background.
- **method** (*str*) – method used to compute enrichment (only 'fisher' is supported currently).

Returns Pandas dataframe with columns: 'terms', 'identifiers', 'foreground', 'background', 'pvalue', 'padj' and 'rejected'.

Example:

```
result = run_regulation_enrichment(regulation_data, annotation, identifier=
↪'identifier', groups=['group1', 'group2'], annotation_col='annotation', reject_
↪col='rejected', group_col='group', method='fisher')
```

analytics_core.analytics.analytics.**run_enrichment**(*data, foreground_id, background_id, annotation_col='annotation', group_col='group', identifier_col='identifier', method='fisher', correction='fdr_bh'*)

Computes enrichment of the foreground relative to a given backgroung, using Fisher's exact test, and corrects for multiple hypothesis testing.

Parameters

- **data** – pandas dataframe with annotations for dataset features (columns: 'annotation', 'identifier', 'source', 'group').
- **foreground_id** (*str*) – group identifier of features that belong to the foreground.

- **background_id** (*str*) – group identifier of features that belong to the background.
- **annotation_col** (*str*) – name of the column containing annotation terms.
- **group_col** (*str*) – name of column containing the group identifiers.
- **identifier_col** (*str*) – name of column containing dependent variables identifiers.
- **method** (*str*) – method used to compute enrichment (only 'fisher' is supported currently).

Returns Pandas dataframe with annotation terms, features, number of foregroung/background features in each term, p-values and corrected p-values (columns: 'terms', 'identifiers', 'foreground', 'background', 'pvalue', 'padj' and 'rejected').

Example:

```
result = run_enrichment(data, foreground='foreground', background='background',␣
↪foreground_pop=len(foreground_list), background_pop=len(background_list),␣
↪annotation_col='annotation', group_col='group', identifier_col='identifier',␣
↪method='fisher')
```

analytics_core.analytics.analytics.**calculate_fold_change**(*df*, *condition1*, *condition2*)

Calculates fold-changes between two groups for all proteins in a dataframe.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns.
- **condition1** (*str*) – identifier of first group.
- **condition2** (*str*) – identifier of second group.

Returns Numpy array.

Example:

```
result = calculate_fold_change(data, 'group1', 'group2')
```

analytics_core.analytics.analytics.**cohen_d**(*df*, *condition1*, *condition2*, *ddof=0*)

Calculates Cohen's d effect size based on the distance between two means, measured in standard deviations. For more information visit https://docs.scipy.org/doc/numpy/reference/generated/numpy.nanstd.html.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns.
- **condition1** (*str*) – identifier of first group.
- **condition2** (*str*) – identifier of second group.
- **ddof** (*int*) – means Delta Degrees of Freedom.

Returns Numpy array.

Example:

```
result = cohen_d(data, 'group1', 'group2', ddof=0)
```

analytics_core.analytics.analytics.**hedges_g**(*df*, *condition1*, *condition2*, *ddof=0*)

Calculates Hedges' g effect size (more accurate for sample sizes below 20 than Cohen's d). For more information visit https://docs.scipy.org/doc/numpy/reference/generated/numpy.nanstd.html.

Parameters

- **df** – pandas dataframe with samples as rows and protein identifiers as columns.

- **condition1** (*str*) – identifier of first group.
- **condition2** (*str*) – identifier of second group.
- **ddof** (*int*) – means Delta Degrees of Freedom.

Returns Numpy array.

Example:

```
result = hedges_g(data, 'group1', 'group2', ddof=0)
```

analytics_core.analytics.analytics.**run_mapper**(*data, lenses=['l2norm'], n_cubes=15, overlap=0.5, n_clusters=3, linkage='complete', affinity='correlation'*)

Parameters

- **data** –
- **lenses** –
- **n_cubes** –
- **overlap** –
- **n_clusters** –
- **linkage** –
- **affinity** –

Returns

analytics_core.analytics.analytics.**run_WGCNA**(*data, drop_cols_exp, drop_cols_cli, RsquaredCut=0.8, networkType='unsigned', minModuleSize=30, deepSplit=2, pamRespectsDendro=False, merge_modules=True, MEDissThres=0.25, verbose=0, sd_cutoff=0*)

Runs an automated weighted gene co-expression network analysis (WGCNA), using input proteomics/transcriptomics/genomics and clinical variables data.

Parameters

- **data** (*dict*) – dictionary of pandas dataframes with processed clinical and experimental datasets
- **drop_cols_exp** (*list*) – column names to be removed from the experimental dataset.
- **drop_cols_cli** (*list*) – column names to be removed from the clinical dataset.
- **RsquaredCut** (*float*) – desired minimum scale free topology fitting index R^2.
- **networkType** (*str*) – network type ('unsigned', 'signed', 'signed hybrid', 'distance').
- **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.
- **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.

Returns Tuple with multiple pandas dataframes.

Example:

```
result = run_WGCNA(data, drop_cols_exp=['subject', 'sample', 'group', 'index'],␣
↪drop_cols_cli=['subject', 'biological_sample', 'group', 'index'], RsquaredCut=0.
↪8, networkType='unsigned', minModuleSize=30, deepSplit=2,␣
↪pamRespectsDendro=False, merge_modules=True, MEDissThres=0.25, verbose=0)
```

analytics_core.analytics.analytics.**most_central_edge**(*G*)

Compute the eigenvector centrality for the graph G, and finds the highest value.

Parameters **G** (*graph*) – networkx graph

Returns Highest eigenvector centrality value.

Return type float

analytics_core.analytics.analytics.**get_louvain_partitions**(*G*, *weight*)

Computes the partition of the graph nodes which maximises the modularity (or try..) using the Louvain heuristices. For more information visit https://python-louvain.readthedocs.io/en/latest/api.html.

Parameters

- **G** (*graph*) – networkx graph which is decomposed.
- **weight** (*str*) – the key in graph to use as weight.

Returns The partition, with communities numbered from 0 to number of communities.

Return type dict

analytics_core.analytics.analytics.**get_network_communities**(*graph*, *args*)

Finds communities in a graph using different methods. For more information on the methods visit:

- https://networkx.github.io/documentation/latest/reference/algorithms/generated/networkx.algorithms.community.modularity_max.greedy_modularity_communities.html
- https://networkx.github.io/documentation/networkx-2.0/reference/algorithms/generated/networkx.algorithms.community.asyn_lpa.asyn_lpa_communities.html
- https://networkx.github.io/documentation/latest/reference/algorithms/generated/networkx.algorithms.community.centrality.girvan_newman.html
- https://networkx.github.io/documentation/latest/reference/generated/networkx.convert_matrix.to_pandas_adjacency.html

Parameters

- **graph** (*graph*) – networkx graph
- **args** (*dict*) – config file arguments

Returns Dictionary of nodes and which community they belong to (from 0 to number of communities).

analytics_core.analytics.analytics.**get_publications_abstracts**(*data, publication_col='publication', join_by=['publication', 'Proteins', 'Diseases'], index='PMID'*)

Accesses NCBI PubMed over the WWW and retrieves the abstracts corresponding to a list of one or more PubMed IDs.

Parameters

- **data** – pandas dataframe of diseases and publications linked to a list of proteins (columns: 'Diseases', 'Proteins', 'linkout' and 'publication').
- **publication_col** (*str*) – column label containing PubMed ids.
- **join_by** (*list*) – column labels to be kept from the input dataframe.
- **index** (*str*) – column label containing PubMed ids from the NCBI retrieved data.

Returns Pandas dataframe with publication information and columns 'PMID', 'abstract', 'authors', 'date', 'journal', 'keywords', 'title', 'url', 'Proteins' and 'Diseases'.

Example:

```
result = get_publications_abstracts(data, publication_col='publication', join_by=[
↪'publication','Proteins','Diseases'], index='PMID')
```

analytics_core.analytics.analytics.**eta_squared**(*aov*)

Calculates the effect size using Eta-squared.

Parameters **aov** – pandas dataframe with anova results from statsmodels.

Returns Pandas dataframe with additional Eta-squared column.

analytics_core.analytics.analytics.**omega_squared**(*aov*)

Calculates the effect size using Omega-squared.

Parameters **aov** – pandas dataframe with anova results from statsmodels.

Returns Pandas dataframe with additional Omega-squared column.

analytics_core.analytics.analytics.**run_two_way_anova**(*df, drop_cols=['sample'], subject='subject', group=['group', 'secondary_group']*)

Run a 2-way ANOVA when data['secondary_group'] is not empty

Parameters

- **df** – processed pandas dataframe with samples as rows, and proteins and groups as columns.
- **drop_cols** (*list*) – column names to drop from dataframe
- **subject** (*str*) – column name containing subject identifiers.
- **group** (*list*) – column names corresponding to independent variable groups

Returns Two dataframes, anova results and residuals.

Example:

```
result = run_two_way_anova(data, drop_cols=['sample'], subject='subject', group=[
↪'group', 'secondary_group'])
```

analytics_core.analytics.analytics.**merge_for_polar**(*regulation_data*, *regulators*, *identifier_col='identifier'*, *group_col='group'*, *theta_col='modifier'*, *aggr_func='mean'*, *normalize=True*)

analytics_core.analytics.analytics.**run_qc_markers_analysis**(*data*, *qc_markers*, *sample_col='sample'*, *group_col='group'*, *drop_cols=['subject']*, *identifier_col='identifier'*, *qcidentifier_col='identifier'*, *qcclass_col='class'*)

analytics_core.analytics.analytics.**run_snf**(*df_dict*, *clusters*, *distance_metric*, *K_affinity*, *mu_affinity*)

Parameters

- **df_dict** –
- **clusters** –

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

WGCNA analytics

analytics_core.analytics.wgcnaAnalysis.**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).

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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).

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

analytics_core.analytics.wgcnaAnalysis.**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.

Vizualization

Viz module

analytics_core.viz.viz.**getPlotTraces**(*data*, *key='full'*, *type='lines'*, *div_factor=10010.0*, *horizontal=False*)

This function returns traces for different kinds of plots.

Parameters

- **data** – Pandas DataFrame with one variable as data.index (i.e. 'x') and all others as columns (i.e. 'y').
- **type** (*str*) – 'lines', 'scaled markers', 'bars'.
- **div_factor** (*float*) – relative size of the markers.
- **horizontal** (*bool*) – bar orientation.

Returns list of traces.

Exmaple 1:

```
result = getPlotTraces(data, key='full', type = 'lines', horizontal=False)
```

Example 2:

```
result = getPlotTraces(data, key='full', type = 'scaled markers', div_
↪factor=float(10^3000), horizontal=True)
```

analytics_core.viz.viz.**get_markdown**(*text*, *args={}*)

Converts a given text into a Dash Markdown component. It includes a syntax for things like bold text and italics, links, inline code snippets, lists, quotes, and more. For more information visit https://dash.plot.ly/dash-core-components/markdown.

Parameters

- **text** (*str*) – markdown string (or array of strings) that adhreres to the CommonMark spec.
- **args** (*dict*) – dictionary with items from https://dash.plot.ly/dash-core-components/markdown.

Returns dash Markdown component.

analytics_core.viz.viz.**get_pieplot**(*data*, *identifier*, *args*)

This function plots a simple Pie plot.

Parameters

- **data** – pandas DataFrame with values to plot as columns and labels as index.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **valueCol** (str) – name of the column with the values to be plotted.
- **textCol** (str) – name of the column containing information for the hoverinfo parameter.
- **height** (str) – height of the plot.
- **width** (str) – width of the plot.

Returns Pieplot figure within the .

analytics_core.viz.viz.**get_distplot**(*data*, *identifier*, *args*)

Parameters

- **data** –
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **group** (str) – name of the column containing the group.

analytics_core.viz.viz.**get_boxplot_grid**(*data*, *identifier*, *args*)

This function plots a boxplot in a grid based on column values.

Parameters

- **data** – pandas DataFrame with columns: 'x' values and 'y' values to plot, 'color' and 'facet' (color and facet can be the same).
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **title** (str) – plot title.
- **x** (str) – name of column with x values.
- **y** (str) – name of column with y values.
- **color** (str) – name of column with colors
- **facet** (str) – name of column specifying grouping
- **height** (str) – plot height.
- **width** (str) – plot width.

Returns boxplot figure within the .

Example:

```
result = get_boxplot_grid(data, identifier='Boxplot', args:{"Title":"Boxplot", 'x
↪':'sample', 'y':'identifier', 'color':'group', 'facet':'qc_class', 'axis':'cols
↪'})
```

analytics_core.viz.viz.**get_barplot**(*data*, *identifier*, *args*)

This function plots a simple barplot.

Parameters

- **data** – pandas DataFrame with three columns: 'name' of the bars, 'x' values and 'y' values to plot.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **title** (str) – plot title.
- **x_title** (str) – plot x axis title.

- **y_title** (str) – plot y axis title.
- **height** (str) – plot height.
- **width** (str) – plot width.

Returns barplot figure within the .

Example:

```
result = get_barplot(data, identifier='barplot', args={'title':'Figure with␣
↪Barplot'})
```

analytics_core.viz.viz.**get_histogram**(*data*, *identifier*, *args*)

Basic histogram figure allows facets cols and rows

param data: pandas dataframe with at least values to be plotted. :param str identifier: id used to identify the div where the figure will be generated. :param ditc args: see below. :Arguments:

- **x** (str) – name of the column containing values to plot in the x axis.
- **y** (str) – name of the column containing values to plot in the y axis (if used).
- **color** (str) – name of the column that defines how the histogram is colored (if used).
- **facet_row** (str) – name of the column to be used as 'facet' row (if used).
- **facet_col** (str) – name of the column to be used as 'facet' column (if used).
- **height** (int) – height of the plot
- **width** (int) – width of the plot
- **title** (str) – plot title.

Returns dash componenet with histogram figure

Example:

```
result = get_histogram(data, identifier='histogram', args={'x':'a', 'color':'group
↪', 'facet_row':'sample', 'title':'Facet Grid Plot'})
```

analytics_core.viz.viz.**get_facet_grid_plot**(*data*, *identifier*, *args*)

This function plots a scatterplot matrix where we can plot one variable against another to form a regular scatter plot, and we can pick a third faceting variable to form panels along the columns to segment the data even further, forming a bunch of vertical panels. For more information visit https://plot.ly/python/facet-trellis/.

Parameters

- **data** – pandas dataframe with format: 'group', 'name', 'type', and 'x' and 'y' values to be plotted.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*ditc*) – see below.

Arguments

- **x** (str) – name of the column containing values to plot in the x axis.
- **y** (str) – name of the column containing values to plot in the y axis.
- **group** (str) – name of the column containing the group.
- **class** (str) – name of the column to be used as 'facet' column.

- **plot_type** (str) – decides the type of plot to appear in the facet grid. The options are 'scatter', 'scattergl', 'histogram', 'bar', and 'box'.
- **title** (str) – plot title.

Returns facet grid figure within the .

Example:

```
result = get_facet_grid_plot(data, identifier='facet_grid', args={'x':'a', 'y':'b
↪', 'group':'group', 'class':'type', 'plot_type':'bar', 'title':'Facet Grid Plot
↪'})
```

analytics_core.viz.viz.**get_ranking_plot**(*data*, *identifier*, *args*)

Creates abundance multiplots (one per sample group).

Parameters

- **data** – long-format pandas dataframe with group as index, 'name' (protein identifiers) and 'y' (LFQ intensities) as columns.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below

Arguments

- **group** (str) – name of the column containing the group.
- **index** (bool) – set to True when multi samples per group. Calculates the mean intensity for each protein in each group.
- **x_title** (str) – title of plot x axis.
- **y_title** (str) – title of plot y axis.
- **title** (str) – plot title.
- **width** (int) – plot width.
- **height** (int) – plot height.
- **annotations** (dict, optional) – dictionary where data points names are the keys and descriptions are the values.

Returns multi abundance plot figure within the .

Example:

```
result = get_ranking_plot(data, identifier='ranking', args={'group':'group',
↪'index':'', 'x_title':'x_axis', 'y_title':'y_axis',
↪        'title':'Ranking Plot', 'width':100, 'height':150, 'annotations':{'GPT~
↪P24298': 'liver disease', 'CP~P00450': 'Wilson disease'}})
```

analytics_core.viz.viz.**get_scatterplot_matrix**(*data*, *identifier*, *args*)

This function pltos a multi scatterplot (one for each unique element in args['group']).

Parameters

- **data** – pandas dataframe with four columns: 'name' of the data points, 'x' and 'y' values to plot, and 'group' they belong to.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below

Arguments

- **group** (str) – name of the column containing the group.
- **title** (str) – plot title.
- **x_title** (str) – plot x axis title.
- **y_title** (str) – plot y axis title.
- **height** (int) – plot height.
- **width** (int) – plot width.
- **annotations** (dict, optional) – dictionary where data points names are the keys and descriptions are the values.

Returns multi scatterplot figure within the .

Example:

```
result = get_scatterplot_matrix(data, identifier='scatter matrix', args={'group':
↪'group', 'title':'Scatter Plot Matrix', 'x_title':'x_axis',
↪                    'y_title':'y_axis', 'height':100, 'width':100, 'annotations
↪':{'GPT~P24298': 'liver disease', 'CP~P00450': 'Wilson disease'}})
```

analytics_core.viz.viz.**get_simple_scatterplot**(*data*, *identifier*, *args*)

Plots a simple scatterplot with the possibility of including in-plot annotations of data points.

Parameters

- **data** – long-format pandas dataframe with columns: 'x' (ranking position), 'group' (original dataframe position), 'name' (protein identifier), 'y' (LFQ intensity), 'symbol' (data point shape) and 'size' (data point size).
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **annotations** (dict) – dictionary where data points names are the keys and descriptions are the values.
- **title** (str) – plot title.
- **x_title** (str) – plot x axis title.
- **y_title** (str) – plot y axis title.
- **height** (int) – plot height.
- **width** (int) – plot width.

Returns annotated scatterplot figure within the .

Example:

```
result = get_scatterplot_matrix(data, identifier='scatter plot', args={
↪'annotations':{'GPT~P24298': 'liver disease', 'CP~P00450': 'Wilson disease'}',
↪                                    'title':'Scatter Plot', 'x_title':'x_axis
↪', 'y_title':'y_axis', 'height':100, 'width':100})
```

analytics_core.viz.viz.**get_scatterplot**(*data*, *identifier*, *args*)

This function plots a simple Scatterplot.

Parameters

- **data** – is a Pandas DataFrame with four columns: "name", x values and y values (provided as variables) to plot.
- **identifier** (*str*) – is the id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **title** (str) – title of the figure.
- **x_title** (str) – plot x axis title.
- **y_title** (str) – plot y axis title.
- **height** (int) – plot height.
- **width** (int) – plot width.

Returns scatterplot figure within the .

Example:

```
result = get_scatteplot(data, identifier='scatter plot', 'title':'Scatter Plot',
↪'x_title':'x_axis', 'y_title':'y_axis', 'height':100, 'width':100}))
```

analytics_core.viz.viz.**get_volcanoplot**(*results*, *args*)

This function plots volcano plots for each internal dictionary in a nested dictionary.

Parameters

- **results** (*dict[dict]*) – nested dictionary with pairwise group comparisons as keys and internal dictionaries containing 'x' (log2FC values), 'y' (-log10 p-values), 'text', 'color', 'pvalue' and 'annotations' (number of hits to be highlighted).
- **args** (*dict*) – see below.

Arguments

- **fc** (float) – fold change threshold.
- **range_x** (list) – list with minimum and maximum values for x axis.
- **range_y** (list) – list with minimum and maximum values for y axis.
- **x_title** (str) – plot x axis title.
- **y_title** (str) – plot y axis title.
- **colorscale** (str) – string for predefined plotly colorscales or dict containing one or more of the keys listed in https://plot.ly/python/reference/#layout-colorscale.
- **showscale** (bool) – determines whether or not a colorbar is displayed for a trace.
- **marker_size** (int) – sets the marker size (in px).

Returns list of volcano plot figures within the .

Example:

```
result = get_volcanoplot(results, args={'fc':2.0, 'range_x':[0, 1], 'range_y':[-1,
↪ 1], 'x_title':'x_axis', 'y_title':'y_title', 'colorscale':'Blues',
↪                    'showscale':True, 'marker_size':7})
```

analytics_core.viz.viz.**run_volcano**(*data*, *identifier*, *args={'alpha': 0.05, 'colorscale': 'Blues', 'fc': 2, 'marker_size': 8, 'num_annotations': 10, 'showscale': False, 'x_title': 'log2FC', 'y_title': '-log10(pvalue)'}*)

This function parsers the regulation data from statistical tests and creates volcano plots for all distinct group comparisons. Significant hits with lowest adjusted p-values are highlighed.

Parameters

- **data** – pandas dataframe with format: 'identifier', 'group1', 'group2', 'mean(group1', 'mean(group2)', 'log2FC', 'std_error', 'tail', 't-statistics', 'padj_THSD', 'effsize', 'efftype', 'FC', 'rejected', 'F-statistics', 'pvalue', 'padj', 'correction', '-log10 pvalue' and 'Method'.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **alpha** (float) – adjusted p-value threshold for significant hits.
- **fc** (float) – fold change threshold.
- **colorscale** (str or dict) – name of predefined plotly colorscale or dictionary containing one or more of the keys listed in https://plot.ly/python/reference/#layout-colorscale.
- **showscale** (bool) – determines whether or not a colorbar is displayed for a trace.
- **marker_size** (int) – sets the marker size (in px).
- **x_title** (str) – plot x axis title.
- **y_title** (str) – plot y axis title.
- **num_annotations** (int) – number of hits to be highlighted (if num_annotations = 10, highlights 10 hits with lowest adjusted p-value).

Returns list of volcano plot figures within the .

Example:

```
result = run_volcano(data, identifier='volvano data', args={'alpha':0.05, 'fc':2.
↪0, 'colorscale':'Blues', 'showscale':False, 'marker_size':6, 'x_title':'log2FC',
↪                          'y_title':'-log10(pvalue)', 'num_annotations':10})
```

analytics_core.viz.viz.**get_heatmapplot**(*data*, *identifier*, *args*)

This function plots a simple Heatmap.

Parameters

- **data** – is a Pandas DataFrame with the shape of the heatmap where index corresponds to rows and column names corresponds to columns, values in the heatmap corresponds to the row values.
- **identifier** (*str*) – is the id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **format** (str) – defines the format of the input dataframe.
- **source** (str) – name of the column containing the source.
- **target** (str) – name of the column containing the target.
- **values** (str) – name of the column containing the values to be plotted.

- **title** (str) – title of the figure.

Returns heatmap figure within the .

Example:

```
result = get_heatmapplot(data, identifier='heatmap', args={'format':'edgelist',
↪'source':'node1', 'target':'node2', 'values':'score', 'title':'Heatmap Plot'})
```

analytics_core.viz.viz.**get_complex_heatmapplot**(*data*, *identifier*, *args*)

analytics_core.viz.viz.**get_notebook_network_pyvis**(*graph*, *args={}*)

This function converts a Networkx graph into a PyVis graph supporting Jupyter notebook embedding.

Parameters

- **graph** (*graph*) – networkX graph.
- **args** (*dict*) – see below.

Arguments

- **height** (int) – network canvas height.
- **width** (int) – network canvas width.

Returns PyVis graph.

Example:

```
result = get_notebook_network_pyvis(graph, args={'height':100, 'width':100})
```

analytics_core.viz.viz.**get_notebook_network_web**(*graph*, *args*)

This function converts a networkX graph into a webweb interactive network in a browser.

Parameters **graph** (*graph*) – networkX graph.

Returns web network.

analytics_core.viz.viz.**network_to_tables**(*graph*)

Creates the graph edge list and node list and returns them as separate Pandas DataFrames.

Parameters **graph** – networkX graph used to construct the Pandas DataFrame.

Returns two Pandas DataFrames.

analytics_core.viz.viz.**generate_configuration_tree**(*report_pipeline*, *dataset_type*)

This function retrieves the analysis pipeline from a dataset .yml file and creates a Cytoscape network, organized hierarchically.

Parameters

- **report_pipeline** (*dict*) – dictionary with dataset type analysis and visualization pipeline (conversion of .yml files to python dictionary).
- **dataset_type** (*str*) – type of dataset ('clinical', 'proteomics', 'DNAseq', 'RNAseq', 'multiomics').

Returns new Dash div with title and Cytoscape network, summarizing analysis pipeline.

analytics_core.viz.viz.**get_network**(*data*, *identifier*, *args*)

This function filters an input dataframe based on a threshold score and builds a cytoscape network. For more information on 'node_size' parameter, visit https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html and https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.eigenvector_centrality_numpy.html.

Parameters

- **data** – long-format pandas dataframe with at least three columns: source node, target node and value (e.g. weight, score).
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **source** (str) – name of the column containing the source.
- **target** (str) – name of the column containing the target.
- **cutoff** (float) – value threshold for network building.
- **cutoff_abs** (bool) – if True will take both positive and negative sides of the cutoff value.
- **values** (str) – name of the column containing the values to be plotted.
- **node_size** (str) – method used to determine node radius ('betweenness', 'ev_centrality', 'degree').
- **title** (str) – plot title.
- **color_weight** (bool) – if True, edges in network are colored red if score > 0 and blue if score < 0.

Returns dictionary with the network in multiple formats: jupyetr-notebook compatible, web brower compatibles, data table, and json.

Example:

```
result = get_network(data, identifier='network', args={'source':'node1', 'target':
↪'node2', 'cutoff':0.5, 'cutoff_abs':True, 'values':'weight',                    ␣
↪         'node_size':'degree', 'title':'Network Figure', 'color_weight': True})
```

analytics_core.viz.viz.**get_network_style**(*node_colors*, *color_edges*)

This function uses a dictionary of nodes and colors and creates a stylesheet and layout for a network.

Parameters

- **node_colors** (*dict*) – dictionary with node names as keys and colors as values.
- **color_edges** (*bool*) – if True, add edge coloring to stylesheet (red for positive width, blue for negative).

Returns stylesheet (list of dictionaries specifying the style for a group of elements, a class of elements, or a single element) and layout (dictionary specifying how the nodes should be positioned on the canvas).

analytics_core.viz.viz.**visualize_notebook_network**(*network*, *notebook_type='jupyter'*, *layout={'height': '700px', 'width': '100%'}*)

This function returns a Cytoscape network visualization for Jupyter notebooks

Parameters

- **network** (*tuple*) – tuple with two dictionaries: network data and stylesheet (see get_network(data, identifier, args)).
- **notebook_type** (*str*) – the type of notebook where the network will be visualized (currently only jupyter notebook is supported)
- **layout** (*dict*) – specific layout properties (see https://dash.plot.ly/cytoscape/layout)

Returns cyjupyter.cytoscape.Cytoscape object

Example:

```
net = get_network(clincorr.dropna(), identifier='corr', args={'source':'node1',
↪'target':'node2',
                                                              'cutoff':0, 'cutoff_abs':True,
                                                              'values':'weight','node_size':
↪'degree',
                                                              'title':'Network Figure',
↪'color_weight': True})
visualize_notebook_network(net['notebook'], notebook_type='jupyter', layout={
↪'width':'100%', 'height':'700px'})
```

analytics_core.viz.viz.**visualize_notebook_path**(*path*, *notebook_type='jupyter'*)

This function returns a Cytoscape network visualization for Jupyter notebooks

Parameters

- **object** (*path*) – dash_html_components object with the cytoscape network (returned by get_cytoscape_network())
- **notebook_type** (*str*) – the type of notebook where the network will be visualized (currently only jupyter notebook is supported)
- **layout** (*dict*) – specific layout properties (see https://dash.plot.ly/cytoscape/layout)

Returns cyjupyter.cytoscape.Cytoscape object

Example:

```
net = get_cytoscape_network(G, identifier='corr', args={'title':'Cytoscape path',
                                                        'stylesheet':stylesheet,
                                                        'layout': layout})
visualize_notebook_path(net, notebook_type='jupyter')
```

analytics_core.viz.viz.**get_pca_plot**(*data*, *identifier*, *args*)

This function creates a pca plot with scores and top "args['loadings']" loadings.

Parameters

- **data** (*tuple*) – tuple with two pandas dataframes: scores and loadings.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below

Arguments

- **loadings** (int) – number of features with highest loading values to be displayed in the pca plot
- **title** (str) – title of the figure
- **x_title** (str) – plot x axis title
- **y_title** (str) – plot y axis title
- **height** (int) – plot height
- **width** (int) – plot width

Returns PCA figure within the .

Example:

```
result = get_pca_plot(data, identifier='pca', args={'loadings':15, 'title':'PCA
↪Plot', 'x_title':'PC1', 'y_title':'PC2', 'height':100, 'width':100})
```

analytics_core.viz.viz.**get_sankey_plot**(*data*, *identifier*, *args={'font': 12, 'height': 800, 'orientation': 'h', 'source': 'source', 'source_colors': 'source_colors', 'target': 'target', 'target_colors': 'target_colors', 'title': 'Sankey plot', 'valueformat': '.0f', 'weight': 'weight', 'width': 800}*)

This function generates a Sankey plot in Plotly.

Parameters

- **data** – Pandas DataFrame with the format: source target weight.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below

Arguments

- **source** (str) – name of the column containing the source
- **target** (str) – name of the column containing the target
- **weight** (str) – name of the column containing the weight
- **source_colors** (str) – name of the column in data that contains the colors of each source item
- **target_colors** (str) – name of the column in data that contains the colors of each target item
- **title** (str) – plot title
- **orientation** (str) – whether to plot horizontal ('h') or vertical ('v')
- **valueformat** (str) – how to show the value ('.0f')
- **width** (int) – plot width
- **height** (int) – plot height
- **font** (int) – font size

Returns dcc.Graph

Example:

```
result = get_sankey_plot(data, identifier='sankeyplot', args={'source':'source',
↪'target':'target', 'weight':'weight','source_colors':'source_colors',
↪                     'target_colors':'target_colors', 'orientation': 'h',
↪'valueformat': '.0f', 'width':800, 'height':800, 'font':12, 'title':'Sankey plot
↪'})
```

analytics_core.viz.viz.**get_table**(*data*, *identifier*, *args*)

This function converts a pandas dataframe into an interactive table for viewing, editing and exploring large datasets. For more information visit https://dash.plot.ly/datatable.

Parameters

- **data** – pandas dataframe.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **title** (*str*) – table title.
- **subset** – selects columns from dataframe to be used. If None, the entire dataframe is used.

Returns new Dash div containing title and interactive table.

Example:

```
result = get_table(data, identifier='table', title='Table Figure', subset = None)
```

analytics_core.viz.viz.**get_multi_table**(*data*, *identifier*, *title*)

analytics_core.viz.viz.**get_violinplot**(*data*, *identifier*, *args*)

This function creates a violin plot for all columns in the input dataframe.

Parameters

- **data** – pandas dataframe with samples as rows and dependent variables as columns.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below

Arguments

- **drop_cols** (list) – column labels to be dropped from the dataframe.
- **group** (str) – name of the column containing the group.

Returns list of violion plots within the .

Example:

```
result = get_violinplot(data, identifier='violinplot, args={'drop_cols':['sample',
↪ 'subject'], 'group':'group'})
```

analytics_core.viz.viz.**create_violinplot**(*df*, *variable*, *group_col='group'*)

This function creates traces for a simple violin plot.

Parameters

- **df** – pandas dataframe with samples as rows and dependent variables as columns.
- **variable** (*(str)*) – name of the column with the dependent variable.

Pram (str) group_col name of the column containing the group.

Returns list of traces to be used as data for plotly figure.

Example:

```
result = create_violinplot(df, 'prptein a', group_col='group')
```

analytics_core.viz.viz.**get_clustergrammer_plot**(*data*, *identifier*, *args*)

This function takes a pandas dataframe, calculates clustering, and generates the visualization json. For more information visit https://github.com/MaayanLab/clustergrammer-py.

Parameters

- **data** – long-format pandas dataframe with columns 'node1' (source), 'node2' (target) and 'weight'
- **identifier** (*str*) – id used to identify the div where the figure will be generated
- **args** (*dict*) – see below

Arguments

- **format** (str) – defines if dataframe needs to be converted from 'edgelist' to matrix
- **title** (str) – plot title

Returns Dash Div with heatmap plot from Clustergrammer web-based tool

analytics_core.viz.viz.**get_parallel_plot**(*data*, *identifier*, *args*)

This function creates a parallel coordinates plot, with sample groups as the different dimensions.

Parameters

- **data** – pandas dataframe with groups as rows and dependent variables as columns.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **group** (str) – name of the column containing the groups.
- **zscore** (bool) – if True, calculates the z score of each values in the row, relative to the row mean and standard deviation.
- **color** (str) – line color.
- **title** (str) – plot title.

Returns parallel plot figure within .

Example:

```
result = get_parallel_plot(data, identifier='parallel plot', args={'group':'group
↪', 'zscore':True, 'color':'blue', 'title':'Parallel Plot'})
```

analytics_core.viz.viz.**get_WGCNAPlots**(*data*, *identifier*)

Takes data from runWGCNA function and builds WGCNA plots.

Parameters

- **data** – tuple with multiple pandas dataframes.
- **identifier** (*str*) – is the id used to identify the div where the figure will be generated.

Returns list of dcc.Graph.

analytics_core.viz.viz.**getMapperFigure**(*data*, *identifier*, *title*)

This function uses the KeplerMapper python package to visualize high-dimensional data and generate a FigureWidget that can be shown or editted. This method is suitable for use in Jupyter notebooks. For more information visit https://kepler-mapper.scikit-tda.org/reference/stubs/kmapper.plotlyviz.plotlyviz.html.

Parameters

- **data** – dictionary. Simplicial complex output from the KeplerMapper map method.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **title** (*str*) – plot title.

Returns plotly FigureWidget within .

analytics_core.viz.viz.**get_2_venn_diagram**(*data*, *identifier*, *cond1*, *cond2*, *args*)

This function extracts the exlusive features in cond1 and cond2 and their common features, and build a two-circle venn diagram.

Parameters

- **data** – pandas dataframe with features as rows and group identifiers as columns.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **cond1** (*str*) – identifier of first group.

- **cond2** (*str*) – identifier of second group.
- **args** (*dict*) – see below.

Arguments

- **colors** (dict) – dictionary with cond1 and cond2 as keys, and color codes as values.
- **title** (str) – plot title.

Returns two-circle venn diagram figure within .

Example:

```
result = get_2_venn_diagram(data, identifier='venn2', cond1='group1', cond2=
↪'group2', args={'color':{'group1':'blue', 'group2':'red'},
↪                'title':'Two-circle Venn diagram'})
```

analytics_core.viz.viz.**plot_2_venn_diagram**(*cond1*, *cond2*, *unique1*, *unique2*, *intersection*, *identifier*, *args*)

This function creates a simple non area-weighted two-circle venn diagram.

Parameters

- **cond1** (*str*) – label of the first circle.
- **cond2** (*str*) – label of the second circle.
- **unique1** (*int*) – number of features exclusive to cond1.
- **unique2** (*int*) – number of features exclusive to cond2.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Parm int intersection number of features common to cond1 and cond2.

Arguments

- **colors** (dict) – dictionary with cond1 and cond2 as keys, and color codes as values.
- **title** (str) – plot title.

Returns two-circle venn diagram figure within .

Example:

```
result = plot_2_venn_diagram(cond1='group1', cond2='group2', unique1=10,␣
↪unique2=15, intersection=8, identifier='vennplot',
↪       args={'color':{'group1':'blue', 'group2':'red'}, 'title':'Two-circle␣
↪Venn diagram'})
```

analytics_core.viz.viz.**get_wordcloud**(*data*, *identifier*, *args={'height': 700, 'margin': 1, 'max_font_size': 100, 'max_words': 400, 'stopwords': [], 'width': 700}*)

This function generates a Wordcloud based on the natural text in a pandas dataframe column.

Parameters

- **data** – pandas dataframe with columns: 'PMID', 'abstract', 'authors', 'date', 'journal', 'keywords', 'title', 'url', 'Proteins', 'Diseases'.
- **identifier** (*str*) – id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **text_col** (str) – name of column containing the natural text used to generate the wordcloud.
- **stopwords** (list) – list of words that will be eliminated.
- **max_words** (int) – maximum number of words.
- **max_font_size** (int) – maximum font size for the largest word.
- **margin** (int) – plot margin size.
- **width** (int) – width of the plot.
- **height** (int) – height of the plot.
- **title** (str) – plot title.

Returns wordcloud figure within .

Example:

```
result = get_wordcloud(data, identifier='wordcloud', args={'stopwords':[
↪'BACKGROUND','CONCLUSION','RESULT','METHOD','CONCLUSIONS','RESULTS','METHODS'],␣
↪                                'max_words': 400, 'max_font_size': 100, 'width
↪':700, 'height':700, 'margin': 1})
```

analytics_core.viz.viz.**get_cytoscape_network**(*net*, *identifier*, *args*)

This function creates a Cytoscpae network in dash. For more information visit https://dash.plot.ly/cytoscape.

Parameters

- **net** (*dict*) – dictionary in which each element (key) is defined by a dictionary with 'id' and 'label' (if it is a node) or 'source', 'target' and 'label' (if it is an edge).
- **identifier** (*str*) – is the id used to identify the div where the figure will be generated.
- **args** (*dict*) – see below.

Arguments

- **title** (str) – title of the figure.
- **stylesheet** (list[dict]) – specifies the style for a group of elements, a class of elements, or a single element (accepts two keys 'selector' and 'style').
- **layout** (dict) – specifies how the nodes should be positioned on the screen.

Returns network figure within .

analytics_core.viz.viz.**save_DASH_plot**(*plot*, *name*, *plot_format='svg'*, *directory='.'*)

This function saves a plotly figure to a specified directory, in a determined format.

Parameters

- **plot** – plotly figure (dictionary with data and layout)
- **name** (*str*) – name of the figure
- **plot_format** (*str*) – suffix of the saved file ('svg', 'pdf', 'png', 'jpeg', 'jpg')
- **directory** (*str*) – folder where figure is to be saved

Returns figure saved in directory

Example:

```
result = save_DASH_plot(plot, name='Plot example', plot_format='svg', directory='/
↪data/plots')
```

analytics_core.viz.viz.**mpl_to_plotly**(*fig*, *ci=True*, *legend=True*)

analytics_core.viz.viz.**get_km_plot**(*data*, *identifier*, *args*)

analytics_core.viz.viz.**get_polar_plot**(*df*, *identifier*, *args*)
This function creates a Polar plot with data aggregated for a given group.

Parameters

- **df** (*dataframe*) – dataframe with the data to plot
- **identifier** (*str*) – identifier to be used in the app
- **args** (*dict*) – dictionary containing the arguments needed to plot the figure (value_col (value to aggregate), group_col (group by), color_col (color by))

Returns Dash Graph

Example:: figure = get_polar_plot(df, identifier='polar', args={'value_col':'intensity', 'group_col':'modifier', 'color_col':'group'})

WGCNA viz

analytics_core.viz.wgcnaFigures.**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.

---

analytics_core.viz.wgcnaFigures.**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.

analytics_core.viz.wgcnaFigures.**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.

analytics_core.viz.wgcnaFigures.**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.

analytics_core.viz.wgcnaFigures.**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.

analytics_core.viz.wgcnaFigures.**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

analytics_core.viz.Dendrogram.**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 analytics_core.viz.Dendrogram.**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

Colors

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

analytics_core.viz.color_list.**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'.

analytics_core.viz.color_list.**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

analytics_core.viz.color_list.**invert_dict**(*d*)

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

d: dict

returns: dict

R wrapper

analytics_core.R_wrapper.**call_Rpackage**(*call='function'*, *designation='aov'*)

analytics_core.R_wrapper.**R_matrix2Py_matrix**(*r_matrix*, *index*, *columns*)

Analytics factory

class analytics_core.analytics_factory.**Analysis**(*identifier*, *analysis_type*, *args*, *data*, *result=None*, *plots={}*)

Bases: object

property identifier

property analysis_type

property args

property data

property result

property plots

property plot

update_plots(*plots*)

generate_result()

get_plot(*name*, *identifier*)

publish_analysis(*directory*)

save_analysis_result(*results_directory*)

save_analysis_plots(*plots_directory*)

make_interactive(*name*, *identifier*)

Utils

analytics_core.utils.**generate_html**(*network*)

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

analytics_core.utils.**append_to_list**(*mylist*, *myappend*)

analytics_core.utils.**neo4j_path_to_networkx**(*paths*, *key='path'*)

analytics_core.utils.**neo4j_schema_to_networkx**(*schema*)

analytics_core.utils.**networkx_to_cytoscape**(*graph*)

analytics_core.utils.**networkx_to_gml**(*graph*, *path*)

analytics_core.utils.**json_network_to_gml**(*graph_json*, *path*)

analytics_core.utils.**json_network_to_networkx**(*graph_json*)

analytics_core.utils.**get_clustergrammer_link**(*net*, *filename=None*)

analytics_core.utils.**generator_to_dict**(*genvar*)

analytics_core.utils.**parse_html**(*html_snippet*)

analytics_core.utils.**convert_html_to_dash**(*el*, *style=None*)

analytics_core.utils.**hex2rgb**(*color*)

analytics_core.utils.**get_rgb_colors**(*n*)

analytics_core.utils.**get_hex_colors**(*n*)

analytics_core.utils.**getMedlineAbstracts**(*idList*)

8.1.5 Notebooks

vis.py

notebooks.development.vis.**vis_network**(*nodes*, *edges*, *physics=False*)

notebooks.development.vis.**draw**(*graph*, *options*, *physics=False*, *limit=100*)

CHAPTER

NINE

PROJECT INFO

9.1 Credits

9.1.1 Development Leads

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

9.1.2 Core Committers

9.1.3 Contributors

9.2 Backers

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

-
-

9.3 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*

9.3.1 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:

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 :)

9.3.2 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:
3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

FINISH THIS PART!!!!!!

4. Create a branch for local development:

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass .....
6. Commit your changes and push your branch to GitHub:
7. Submit a pull request through the GitHub website.

9.3.3 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.

9.3.4 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*.

9.4 History

9.4.1 1.0b0 (2020-05-11)

- First release on GitHub.

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

9.5 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

TEN

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

g

n

r