Clinical Knowledge Graph Documentation

Release 1.0.0

Alberto Santos, Ana Rita Colaço, Annelaura B. Nielsen

Jan 16, 2020

CONTENTS

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

CHAPTER

ONE

CLINICAL KNOWLEDGE GRAPH

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

1.1 Abstract

Several omics data types are already used as diagnostic markers, beacons of possible treatment or prognosis. Advances in technology have paved the way for omics to move into the clinic by generating increasingly larger amounts of high-quality quantitative and qualitative data. Additionally, knowledge around these data has been collected in diverse public resources, which has facilitated the understanding of these data to some extent. However, there are several challenges that hinder the translation of high-throughput omics data into identifiable, interpretable and actionable clinical markers. One of the main challenges is the interpretation of the multiple hits identified in these experiments. Furthermore, a single omics dimension is often not sufficient to capture the full complexity of disease, which would be aided by integration of several of them. To overcome these challenges, we propose a system that integrates multi-omics data and information spread across a myriad of biomedical databases into a Clinical Knowledge Graph (CKG). This graph focuses on the data points or entities not as silos but as related components of a graph. To illustrate, in our system an identified protein in a proteomics experiment encompasses also all its related components (other proteins, diseases, drugs, etc.) and their relationships. Thus, our CKG facilitates the interpretation of data and the inference of meaning by providing relevant biological context. Further, ~. Here we describe the current state of the system and depict its use by applying it to use cases such as treatment decisions using cancer genomics and proteomics.

1.2 Features

- Cross-platform: Windows, Mac, and Linux are officially supported.

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*

2.1 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 "bioinfo1112".

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 redy go to!

2.1.1 Add Neo4j graph database to *.bashrc*

In order run the graph database, add the path to the database to your `.bashrc` (or `.bash_profile`) file.

To find out which of the files your machine uses, go to the terminal and type `more ~/.bash` and double press the tab key on your keyboard for auto-complete. Immediately below, multiple filenames will be printed, check if among those, is `.bashrc` or `.bash_profile`.

Note: The bash file can be name `.bashrc` or `.bash_profile`. if your system does not have either, created one of them (e.g. vi ~/.bash_profile).

1. Open the `.bash_profile` (or `.bashrc`) with your favourite text editor. In this case, we use the **vi** editor:

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

Note: To edit with **vi** press `i` on your keyboard.

2. Add the path to the previously created Neo4j database to the file:

```
$ NEO4J_HOME="/Users/username/Library/Application Support/Neo4j Desktop/Application/neo4jDatabases/da
$ export NEO4J_HOME
```

Note: To save and close a file with **vi** editor, press `Esc` followed by `:wq`.

Warning: Depending on your system, the path may vary. To check the path to the database go to `Logs` in the Neo4j Desktop interface.

3. Reload the `.bashrc` (or `.bash_profile`) file:

```
$ source ~/.bashrc
```

or

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

2.2 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 --version
```

And where this Python version is:

```
$ which python3
```

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
```

2. Add at the end of the file:

3. Save and close the bash file

4. Make the alias available in the current session:

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

Note: If you don't have **Python 3.6** installed, go to `pyhton.org` and 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`.

2.2.1 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 costum 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
```

4. Activate the virtual environment by running:

```
$ source 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
```

2.2.2 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 neccessary to run the Clinical Knowledge graph in `requirements.txt`. To install all the packages required, simply run:

```
$ pip install -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:

```
$ cd CKG/
$ 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.

2.2.3 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/".

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

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

2.2.4 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 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* and *DrugBank Vocabulary*.
- SNOMED-CT: *Download SNOMED CT to ICD-10-CM Mapping Resources*.

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.

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`

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 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 aproximately 10 hours but depending on a multitude of factors, it can take up to 16 hours.

From a backup dump file

Alternatively, you can use the available dump file and load the graph database contained in it:

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

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/2019-11-04.dump --database=graph.db --force
```

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

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

```
$ sudo bin/neo4j-admin load --from=backups/graph.db/2019-11-04.dump --database=graph.db --force
$ sudo chown -R username data/databases/graph.db/
```

Once you are done, restart the database and you are good to go!

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 analysis.

3.1.1 Py2neo connector

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.

Python prompt from graphdb_connector import connector driver = connector.getGraphDatabaseConnectionConfiguration()

Once the connection is established, we can start querying the database. For example:

Python prompt 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.

3.2 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 graphdb_builder/builder
$ python create_user.py -u username -d password -n name -e email -s second_email -p phone_number -a a
```

From an excel file: *(multiple users)*

```
$ cd 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
```

For each new user, an access password is set be the same the the `username`. All users are advised to modify the default password as soon as they are granted access.

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)

2. In a separate terminal window:

   - Navigate to `report_manager`

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

   - Start a celery queue from the report_manager directory:

```
$ celery -A worker worker -l debug
```

3. In third 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 you web browser.
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). After all fields are filled in, please revise all the information and press `Create Project`.

The page will refresh and once finished, the project identifier will be depicted in front of the `Project information` header. 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".

Fill in the ExperimentalDesign 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 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'_**.

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.

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").

3.4 Upload project experimental data

3.4.1 Prepare data for upload

Clinical Data

Open the Clinical Data excel file, automatically download 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.
- **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 be identical to the disease 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.
- **biological_sample quantity**: Amount of biological sampl.
- **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.
- **analytical_sample quantity**: Amount of sample used in the experiment.
- **analytical_sample quantity_units**: Unit.
- **grouping1**: Annotate grouping of each sample.
- **grouping2**: If there are more than one grouping (two independent variables) use this column to add a second level.

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

Additional clinical information about your study subjects can be added in the subsequent columns. Please use SNOWMED 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 variavles, please visit the SNOMED browser.

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 brackets: 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.

Proteomics data Do not perform any imputations or similar on your data before uploading it. This will be carried out by the CKG.

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

3.4.2 Additional Clinical Data columns

- **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.

3.4.3 Upload data

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

Go to **'http://localhost:5000/apps/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'_**.
2. Select the type of data you will upload first
   - If `proteomics` or `longitudinal_proteomics` is selected, please also select the processing tool used (`MaxQuant` or `Spectronaut`).
3. Drag and drop or select the files to upload to the selected data type.
   - Multiple files can be selected at once. This is specially important in the case of proteomics files, please make sure you select all of the relevant MS files at once.
4. Click `Upload Data`.
5. Select another data type to upload, drag and drop or select the files to upload, and click `Upload Data`.
6. When you have uploaded all the relevant files, click `Add to CKG`. After the `Add to CKG` is clicked, it will be deactivated. To restore its function, refresh the page.
7. Once the data is uploaded, 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:`

Note: To replace the files uploaded, just select the correct data type and processing tool, reselect the files and click `Upload Data` again.

Warning: Make sure the correct data type and processing tool (when applicable) are selected before selecting files.

3.5 Define data analysis parameters

3.5.1 Clinical data analysis parameters

3.5.2 Proteomics data analysis parameters

3.5.3 Whole Exome Sequencing data analysis parameters

3.5.4 Multiomics data analysis parameters

3.6 Accessing the analysis report

3.6.1 Report: Dash web app

3.6.2 Report: Jupyter-notebook

3.7 Report 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 FEATUES

- **CKG Statistics**: *Imports stats* | *Graph database stats*
- **Jupyter notebooks**: *Notebooks*
- **Retrieving data from the CKG**: *DB Querying* >>>>> query_utils.find_queries_involving_nodes + query_utils.read_knowledge_queries()
- **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, vizualize 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 offical 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 visualization 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 inplace 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 referes to Jupyter notebooks that have been finished and properly tested by the devleopers, 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 visualization of all the plots and tables that constitue 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 visualization.

- **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 costum 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 visualization 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 diferent 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) controlos 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 prioritize 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 Standardizing the data analysis

6.6 Visualization plots

6.7 Python - R interface

CHAPTER

SEVEN

SYSTEM REQUIREMENTS

- **Mac OS X**: *Requirements*
- **Linux**: *Requirements*
- **Windows**: *Requirements*

7.1 Mac OS X requirements

7.2 Linux requirements

7.3 Windows requirements

CHAPTER

EIGHT

API REFERENCE

8.1 Clinical Knowledge Graph package

`graphdb_connector`
`graphdb_builder`
`report_manager`
`analytics_core`
`notebooks`

8.1.1 Graph Database Connector

connector.py

`graphdb_connector.connector.`**`getGraphDatabaseConnectionConfiguration`**(*configuration=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*)

8.1.2 Graph Database Builder

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.

Parameters **stats** (*list*) – a list with statistics collected from each importer function.

Returns 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.

Parameters

- **statsDf** – a pandas dataframe with the new statistics from the importing.
- **statsName** (*str*) – 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*`)`

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*`)`

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.

8.1.3 Report Manager

Report Queries

Query Utils

Report Dash Apps

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

`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 <div id="_dash-app-content">.

`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 <div id="_dash-app-content">, 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 <div id="_dash-app-content">.

`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 <div id="_dash-app-content">.

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 (CPxxxxxxxxxxxx).

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 (CPxxxxxxxxxxxx).
- **`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.

Data Upload

dataset.py

`class` `report_manager.dataset.`**`Dataset`**(*identifier*, *dataset_type*, *configuration=None*, *data={}*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: `object`

`property identifier`

`property dataset_type`

`property data`

`property analyses`

`property configuration`

`property analysis_queries`

`property report`

`generate_dataset`()

`update_report`(*report*)

`update_analysis_queries`(*query*)

`get_dataframe`(*dataset_name*)

`get_dataframes`(*dataset_names*)

`list_dataframes`()

`get_analysis`(*analysis*)

`update_data`(*new*)

`update_analyses`(*new*)

`set_configuration_from_file`(*configuration_file*)

`update_configuration_from_file`(*configuration_file*)

`get_dataset_data_directory`(*directory='../../../data/reports'*)

`query_data`()

`send_query`(*query*)

extract_configuration(*configuration*)

add_configuration_to_report(*report_pipeline*)

generate_report()

save_dataset_recursively(*dset*, *group*, *dt*)

save_dataset(*dataset_directory*)

save_dataset_recursively_to_file(*dset*, *dataset_directory*, *base_name=''*)

save_dataset_to_file(*dataset_directory*)

save_report(*dataset_directory*)

load_dataset_recursively(*dset*, *loaded_dset={}*)

load_dataset(*dataset_directory*)

load_dataset_report(*report_dir*)

generate_knowledge()

class report_manager.dataset.**MultiOmicsDataset**(*identifier*, *data*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.Dataset*

get_dataframes(*datasets*)

generate_knowledge()

class report_manager.dataset.**ProteomicsDataset**(*identifier*, *dataset_type='proteomics'*, *data={}*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.Dataset*

generate_dataset()

process_dataset()

processing()

generate_knowledge()

class report_manager.dataset.**LongitudinalProteomicsDataset**(*identifier*, *data={}*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.ProteomicsDataset*

class report_manager.dataset.**ClinicalDataset**(*identifier*, *data={}*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.Dataset*

generate_dataset()

process_dataset()

processing()

generate_knowledge()

class report_manager.dataset.**DNAseqDataset**(*identifier*, *dataset_type*, *data={}*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.Dataset*

generate_dataset()

class report_manager.dataset.**RNAseqDataset**(*identifier*, *data={}*, *configuration=None*, *analyses={}*, *analysis_queries={}*, *report=None*)

Bases: *report_manager.dataset.Dataset*

generate_dataset()

knowledge.py

class report_manager.knowledge.**Knowledge**(*identifier*, *data*, *nodes={}*, *relationships={}*, *queries_file=None*, *colors={}*, *graph=None*, *report={}*)

Bases: object

property identifier

property data

property entities

property nodes

property relationships

property queries_file

property colors

property default_color

property report

property graph

generate_knowledge_from_regulation(*entity*)

genreate_knowledge_from_correlation(*entity_node1*, *entity_node2*, *filter*, *cutoff=0.5*)

generate_knowledge_from_wgcna(*data*, *entity1*, *entity2*, *cutoff=0.2*)

generate_knowledge_from_annotations(*entity1*, *entity2*, *filter=None*)

generate_knowledge_from_similarity(*entity='Project'*)

generate_knowledge_from_queries(*entity*, *queries_results*)

send_query(*query*)

query_data(*replace*)

generate_cypher_nodes_list()

generate_knowledge_graph()

reduce_to_subgraph(*nodes*)

get_knowledge_graph_plot()

generate_report(*visualization='sankey'*)

save_report(*directory*)

class report_manager.knowledge.**ProjectKnowledge**(*identifier*, *data*, *nodes={}*, *relationships={}*, *colors={}*, *graph=None*, *report={}*)

Bases: *report_manager.knowledge.Knowledge*

generate_knowledge()

class report_manager.knowledge.**ProteomicsKnowledge**(*identifier*, *data*, *nodes={}*, *relationships={}*, *colors={}*, *graph=None*, *report={}*)

Bases: *report_manager.knowledge.Knowledge*

generate_knowledge()

class report_manager.knowledge.**ClinicalKnowledge**(*identifier*, *data*, *nodes={}*, *relationships={}*, *colors={}*, *graph=None*, *report={}*)

Bases: *report_manager.knowledge.Knowledge*

generate_knowledge()

class report_manager.knowledge.**MultiOmicsKnowledge**(*identifier*, *data*, *nodes={}*, *relationships={}*, *colors={}*, *graph=None*, *report={}*)

Bases: *report_manager.knowledge.Knowledge*

generate_knowledge()

project.py

class report_manager.project.**Project**(*identifier*, *configuration_files={}*, *datasets={}*, *knowledge=None*, *report={}*)

Bases: object

A project class that defines an experimental project. A project can be of different types, contain several datasets and reports.

Example:

```
p = Project(identifier="P0000001", datasets=None, report=None)
p.show_report(environment="notebook")
```

property identifier

property configuration_files

property queries_file

property name

property acronym

property data_types

append_data_type(*data_type*)

property responsible

property description

property status

property num_subjects

property datasets

property knowledge

property report

property similar_projects

property overlap

get_dataset(*dataset*)

update_dataset(*dataset*)

update_report(*new*)

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

get_report_directory()

get_downloads_directory()

set_attributes(*project_info*)

from_dict(*attributes*)

to_dict()

to_dataframe()

list_datasets()

to_json()

from_json(*json_str*)

query_data()

check_report_exists()

load_project_report()

load_project(*directory*)

load_project_data()

build_project(*force=False*)

get_projects_overlap(*project_info*)

get_similar_projects(*project_info*)

generate_project_attributes_plot()

generate_project_similarity_plots()

generate_overlap_plots()

get_similarity_network_style()

get_similarity_network()

generate_knowledge()

generate_project_info_report()

generate_report()

notify_project_ready(*message_type='slack'*)

empty_report()

save_project_report()

save_project_datasets_reports()

save_project()

save_project_datasets_data()

show_report(*environment*)

download_project()

download_project_report()

download_knowledge(*directory*)

download_project_datasets()

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.**is_jsonable**(*x*)

report_manager.utils.**convert_dash_to_json**(*dash_object*)

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*)

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

Bases: `json.encoder.JSONEncoder`

default(*obj*)

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

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

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

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

Bases: `json.encoder.JSONEncoder`

default(*obj*)

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

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

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
```

(continues on next page)

(continued from previous page)

```
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

worker.py

8.1.4 Analytics Core

Analysis

Analytics

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'])
```

(continues on next page)

(continued from previous page)

```
```

`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'*)

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.

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.`**`polish_median_normalization`**(*data, max_iter=10*)

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 = polish_median_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 rows and samples as columns.

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*)

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.**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'*)

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', 'distribiution', 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.

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_normality**(*data*)

Checks whether columns (features) in the provided matrix follow a normal distribution (Requires at least 8 rows). Uses Pigouin test Normality: https://pingouin-stats.org/generated/pingouin.normality.html

Parameters **data** – pandas dataframe.

Returns a pandas data frame with features as rows, test statistic, pvalue, true/false normally distributed. None if

number of rows below 8.

Example:

```
result = check_normality(data)
```

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_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', 'indep')*)

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** (*tuple*) – first string corresponds to FDR correction type ('fdr', '2fdr'), and second string determines which method to use (fdr:'indep', 'negcorr', 2fdr:'bky', 'bh').

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', 'indep'))
```

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

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

Parameters

- **df** (*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** (*tuple*) – first string corresponds to FDR correction type ('fdr', '2fdr'), and second string determines which method to use (fdr:'indep', 'negcorr', 2fdr:'bky', 'bh').

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

Example:

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

`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', 'indep')*)

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** (*tuple*) – first string corresponds to FDR correction type ('fdr', '2fdr'), and second string determines which method to use (fdr:'indep', 'negcorr', 2fdr:'bky', 'bh').

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',
↪'indep'))
```

`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_paired_ttest**(*df*, *condition1*, *condition2*)

Calculates the t-test on RELATED samples belonging to two different groups. For more information visit https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.ttest_rel.html.

Parameters

- **df** – pandas dataframe with groups and subjects as rows and protein identifier as column.
- **condition1** (*str*) – identifier of first group.
- **condition2** (*str*) – identifier of second group.

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

Example:

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

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*)

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.

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*, *group='group'*, *alpha=0.05*)

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 as rows and protein identifier as column
- **group** (*str*) – column label containing the within factor
- **alpha** (*float*) – significance level

Returns Pandas dataframe.

Example:

```
result = calculate_THSD(df, group='group', alpha=0.05)
```

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

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*)

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*, *group='group'*)

Calculates one-way ANOVA using scipy stats.

Parameters

- **`df`** – pandas dataframe with group as rows and protein identifier as column
- **`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=50*)

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.

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(group2)', 'Log2FC', 'std_error', 'tail', 't-statistics', 'padj_THSD', '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.**run_repeated_measurements_anova**(*df, alpha=0.05, drop_cols=['sample'], subject='subject', group='group', permutations=50*)

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, group, permutations, alpha, max_permutations*)

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
- **max_permutations** (*int*) – maximum number of permutations according to number of samples

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='indep', permutations=50*)

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 for false discovery rate ('indep', 'negcorr')
- **alpha** (*float*) – error rate for multiple hypothesis correction
- **permutations** (*int*) – number of permutations used to estimate false discovery rates.

Returns Pandas dataframe with columns 'identifier', 'group1', 'group2', 'mean(group1)', 'mean(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='indep', permutations=50)
```

`analytics_core.analytics.analytics.`**`run_samr`**(*df, subject='subject', group='group', drop_cols=['subject', 'sample'], alpha=0.05, s0=1, permutations=250*)

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.

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.

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)
```

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_regulation_enrichment**(*regulation_data, annotation, identifier='identifier', groups=['group1', 'group2'], annotation_col='annotation', reject_col='rejected', group_col='group', method='fisher'*)

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*, *background*, *foreground_pop*, *background_pop*, *annotation_col='annotation'*, *group_col='group'*, *identifier_col='identifier'*, *method='fisher'*)

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`** (*`str`*) – group identifier of features that belong to the foreground.
- **`background`** (*`str`*) – group identifier of features that belong to the background.
- **`foreground_pop`** (*`int`*) – number of features in the foreground population.
- **`background_pop`** (*`int`*) – number of features in the background population.
- **`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*)

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.`**`run_snf`**(*df_dict, clusters, distance_metric, K_affinity, mu_affinity*)

Parameters

- **`df_dict`** –
- **`clusters`** –

WGCNA analytics

`analytics_core.analytics.wgcnaAnalysis.`**`get_data`**(*data, drop_cols_exp=['subject', 'group', 'sample', 'index'], drop_cols_cli=['subject', 'group', 'biological_sample', 'index']*)

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 <div id="_dash-app-content">.

`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_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 <div id="_dash-app-content">.

Example:

```
result = get_barplot(data, identifier='barplot', args={'title':'Figure with␣
↪Barplot'})
```

`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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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_scatterplot_matrix_old**(*data*, *identifier*, *args*)

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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_old**(*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.**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 <div id="_dash-app-content">.

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, title, colors=('#C2D4FF', '#F5F8FF'), subset=None, plot_attr={'font': 12, 'height': 2500, 'width': 1500}, subplot=False*)

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content"> .

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 <div id="_dash-app-content"> .

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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 <div id="_dash-app-content">.

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')
```

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*)

Bases: `object`

property identifier

property analysis_type

property args

property data

property result

`generate_result`()

`get_plot`(*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.`**`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.`**`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:

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

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

Submit Feedback

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

If you are proposing a feature:

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

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:

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

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv CKG
$ cd CKG/
```

FINISH THIS PART!!!!!!

4. Create a branch for local development:

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

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass . . . . . .
6. Commit your changes and push your branch to GitHub:

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

7. Submit a pull request through the GitHub website.

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 0.1.0 (2019-12-10)

- First release on GitHub.

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