Project Overview

We have designed a website to work with two datasets .

The website is composed by three parts which interact to analyze, convert the data in a usable format and present the results to the user. They are respectively "functions.py", "mediator.py" and "website.py".

Following the specification requested on the goal of the project, Part 2, **"functions.py"**, is the core of the analysis and contains all the classes and methods needed in order to perform the operations on the datasets. Part3 is **"website.py"** and contains all the functions needed to interact with the user through a webpage and present him the results. While part 1 is **"mediator.py"** and is the component which links the other two parts and allows them to communicate.

In general, **"functions.py"** returns the results as **pandas.DataFrame**, and **"mediator.py"** converts them in a list which is added to a dictionary containing also information on the length of the table and on the labels.

```
data={
'nrows':'n of rows in the table',
'labels':'labels of the table',
'rows':'rows of the table'
}
```

Download as .pdf!

Features

Here are described all the features and techniques used to write the program.

The website is made using bulma as CSS framework and FontAwesome for the icons.

Templates

To keep the style of the website consistent throughout all the webpages, we used a template called **'base.html'** which contains code needed on all webpages like:

Initialization code: here is declared the doctype, the language, the stylesheets, the icons and jsquery.

Navigation-bar: here is all the code pertaining the navigation-bar at the top of the webpage with its links.

Popup notification: in case there is a message that needs to be communicated to the user, here is the code which lets a popup notification appear at the top of the screen. All the messages can be sent by using `flash()` in a python file with Flask, they will queue up until a new page is loaded, then they will be presented. As argument it needs something to pass to the html file like a string or a dictionary. We've set up the popup to so that it's possible to pass a dict to provide some degree of customization and let the message box be usable both for warnings and for information.

```
message = {
'type':'colour of the popup',
'header':'title of the popup',
'message':'text of the popup',
'details':'additional details'
} .
```

The `'type'` can be of the all colors possible with bulma-message. You just need to write the actual type, without `is-`.

Example: `info` , `success` or `warning` .

Info

Lorem ipsum dolor sit amet, consectetur adipiscing elit. **Pellentesque risus mi**, tempus quis placerat ut, porta nec nulla.

Success

Lorem ipsum dolor sit amet, consectetur adipiscing elit. **Pellentesque risus mi**, tempus quis placerat ut, porta nec nulla.

Warning

Lorem ipsum dolor sit amet, consectetur adipiscing elit. **Pellentesque risus mi**, tempus quis placerat ut, porta nec nulla.

Jinja block

We used `{% block name %}{% endblock %}` inside the html files to pass pieces of code to "base.html" from which all html files inherit from. In this way it's possible to personalize the code for every webpage, for example by passing the code for the body using `{% block content %} {% endblock %}` .

Macros

To avoid rewriting many times over the same code, we have implemented the use of macros to be called when needed.

For example in all the html documents of the operations, to show the tables, there are macros with the style customizable when called. The macros used are 3: **mytable, mytable_info, mytable_with_pmid_links**.
The first one is the main one, mytable_info instead is used in the webpage '/info' to show head and tail of the two datasets, while the last one is used to add links to the original articles of pmid in the tables.

To let one macro be usable for all the tables and operations, the data passed to the html file by jinja and used by the tables needs to be as general as possible, thus is written as a dictionary with the following keys:

```
data = {
'labels': 'labels of the table',
'rows': 'rows of the table',
'length': 'length of the table'
}
```

Download button

In many pages there is the possibility to download something by clicking a button. To make it as clean and general as possible, we have implemented the "download()" function inside in "website.py". It allows to download both an existing file and a table with the results of some operation done on the datasets.

To achieve this, we leveraged the power of flask-caching to save the table in the cache to be later retrieved by "download()" which then converts it to a tsv file and lets the user download it.

Example: Suppose that the user wants to find the evidences in literature of the relation between the gene "ACE2" and COVID-19. Here's what happens:
Step1) "geneEvidences()" inside "website.py" is called, and it asks the user for a gene.
Step 2) "geneEvidences()" then sends the gene to "mediator.py" which returns back the results.
Step 3) Then "geneEvidences()" render the html page with the table and a the name of the file as arguments. In our example the gene is "ACE2" and the function is "geneEvidences" thus the name of the file will be "ACE2_evidences"
Step 4) The html file then shows the table and when the download-button is clicked, the name of the file (that will be the name of the .tsv table) is passed to "download()" in "website.py". .
Step 5) Then "download()" retrieves the name and checks if it corresponds to an already existing file. If it does then the file is downloaded. In our example on the other hand it doesn't, thus it retrieves the table from the cache, converts [*] it to a .tsv file and returns it to be downloaded.

Suppose instead that the user is in the homepage and clicks on the "gene_evidences.tsv" button which allows him to download the whole dataset. "Homepage.html" will send to download the relative path: "dataset/gene_evidences.tsv". "download()" then search if the file exists, and given that it does it will let the user download it.

Going more in detail, the cache saves the data in a dictionary-like style, this means that each object is saved in a "key:value" pair, and the key for the table is "name_file", the same name by which the table will be saved as .tsv.

[*] To know more about the process of conversion from the table dictionary to a tsv file please refer to the function download

Documentation

On the website it's also possible to view the documentation, as you're doing now. To provide a better readability, here you have a side bar with the links accessible inside the documentation.

Every page of the documentation use "docs_base.html" as a template which extends "base.html".

Given that every webpage of the documentation follow the same format, we have created some macros inside "docs_macros.html" to show functions, classes and methods in every webpage. The function "getDocumentation()" inside "mediator.py" loads the .json files containing all the text and converts them to a dictionary. It finally returns it to "documentation()" inside "website.py".

"documentation()" inside "website.py" manages all the documentation webpages, without the need to write one function for every webpage of the documentation.

Browse Datasets

In the homepage you have the possibility to browse through the datasets divided in pages to allow for better readability. The pagination is performed inside "browseGeneDataset" and "browseDiseaseDataset" in "website.py" by flask-paginate. Every entry of the dataset shown in the table has also a link to the its publication on PubMed.