Programming project: DisGeNET Web Application

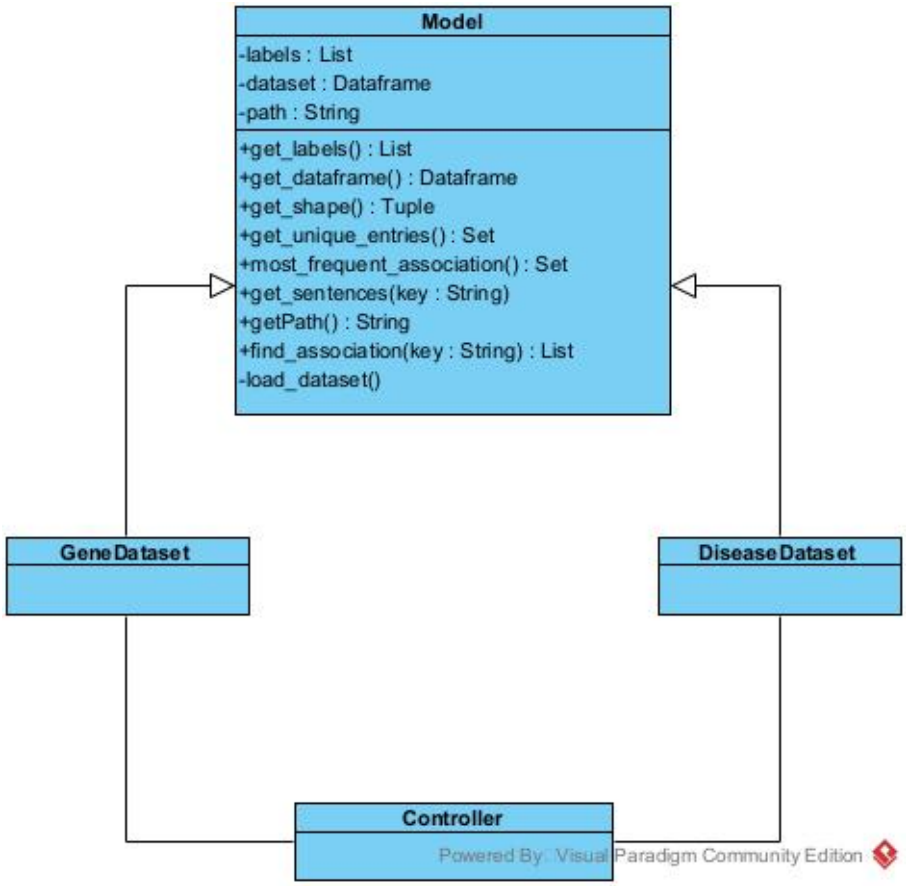
Goal

The goal of this project is to create a Web Application using **Python**: in particular the application should follow the Object Oriented Paradigm.
Flask is the web framework used for the development of a web interface while data parsing and handling was done by using **Pandas**, **Numpy** and custom made classes.
The interface the user sees should have all the operations required which are:

- Get the **shape** of the dataset.
- Get the **labels** of the dataset.
- Get the **unique entries** of genes or diseases in an ordered manner (ascending order).
- Get the **sentences in literature** associated to a gene or disease given by the user.
- Get the **top 10 most frequent distinct associations** between genes and diseases.
- Get the list of gene/disease a gene/disease is **associated** with based on the user input.

UML Diagram

An in-depth explanation of each class is provided also using CRC cards, the only exception in this Class Diagram is the Controller block, which is not a class but just a python module.
Due to its importance in the program flow we thought it was important to underline its presence in the class diagram.



Class Model

Model GeneDataset, DiseaseDataset	
• Getting labels of the dataframe • Elaborating dataframe • Recognising the path of stored datasets • Getting the shape of the dataframe • Loading the dataset • Merging the dataframes • Recording the number of different genes/diseases in ascending order • Provide the list of sentences correlated to a specific gene/disease, given a gene symbol/disease name or its Id • Record the 10 most frequent distinct associations between genes and diseases • Provide list of associations between genes and diseases	

- GETTER AND SETTER

- Def get_labels():

  This function allows us to get the set of labels of the DataFrame that we are considering.

  In particular it returns a set with the labels of the columns of the DataFrame.

- Def get_dataframe():

  This function allows us to work with the DataFrame, using *self.__dataframe* we access the DataFrame from Pandas.

  Using this function we are able to perform several operations and methods on the DataFrame we are taking into account.

- Def get_path():

  Returns the path of the dataset of the model, given that there is one. An empty string is returned if no path is present.

- Def get_shape():

  This function returns the shape of the DataFrame as a tuple: (rows, columns).

  We use the .shape() provided by pandas to return it.

- CONSTRUCTOR

- Def __init__():

  This is the default constructor of the class, the arguments of the method depend on which way the dataset is retreived: if the user already has a pandas DataFrame ready to be used as a model then by setting the *is_dataframe* flag to True the DataFrame can be directly passed using the *df* argument; otherwise to load a DataFrame (which is expected to be formatted as a tsv file) a path needs to be passed, the *is_dataframe* flag is by default set to False and the *__load_dataset()* function is called to perform the parsing operation.

- METHODS

- Def get_unique_entries():

  This function provides the list of unique entries of the DataFrame sorted in an ordered way.

  The function has one argument: *key* that represents one of the labels.

  Firstly, we check if the *key* is in the set of *labels*, if this condition is True it returns a sorted Numpy Array that includes all the elements of the selected column without repetitions.

  If on the other hand, the inserted key doesn't belong to the set the function returns an empty list.

- Def get_sentences():

  In the given DataFrames one of the columns is constituted by sentences taken from the articles. This function allows the user to get a list of sentences correlated to a specific element of a given column.

  Actually, the function takes two parameters: *key* that represents one of the labels, *attribute* that represents one of the elements of the chosen key.

  Firstly, we check if the *key* belongs to the labels, if this condition is True we check if the *attribute* belongs to the list returned by the *get_unique_entries(key)* method. If the condition is True the function creates a list with the sentences related to the chosen attribute and returns it.

  If the conditions are not satisfied it prints an error message and returns None.

- Def __load_dataset():

  This function is used to load a tsv dataset as a pandas dataframe.

  To avoid any error that could stop the execution of the server a try .. except construct is used.

  In the try block the read_csv function pandas offers is used to load the data, if the execution goes through the labels are assigned using the .columns attribute pandas dataframes have.

In case of a failed execution the __dataframe is set to None and a string is printed out as a way to tell the user that something went wrong.

- Def find_association():

  Using the *merge_models()* method we obtain a merged DataFrame which is used by our function in order to provide a list of associated elements. Given the element of a column the function returns a list of associated attributes of another column.

  The function takes four parameters: *key* that represents one of the labels, *attribute* that represents one of the elements of the chosen key, *merged_model* that represents the merged DataFrame, *second_key* that represents the other label.

  Firstly, we check if the *key* belongs to the labels, if this condition is True we check if the *attribute* belongs to the list returned by the *get_unique_entries(key)* method. If the condition is True the function creates a set with all the elements of the *second_key* column that are associated with the elements of the first *key*.

  If one of the conditions is not satisfied the function returns None and prints an error message.

- STATICMETHODS

- Def merge_models():

  We implemented this function as a *@staticmethod*. This is a type of method that is bound to the class rather than is object and it doesn't require a class instance creation. So, it knows nothing about the class and just deals with the parameters and for this reason it is independent from the state of the object.

  The method is able to merge two different DataFrame utilizing one common element.

  The function takes three parameters: *m1* which is an instance of the class Model and represents the first data set, *m2* which is an instance of the class Model and represents the second data set, *on_key* that represents the common label.

  Firstly, we check if the chosen label belongs to both DataFrames, if the condition is True, using the Pandas method *.merge()* associated with the parameter *"inner"* (the inner joint of the DataFrames) the function returns an instance of the class Model where the flag *is_dataframe*=True and the attribute *df* is the newly created merged DataFrame.

- Def most_frequent_association():

  This is also a *@staticmethod* that provides the ten top most frequent associations between the two DataFrames "genes" and "diseases".

  The function takes three parameters: *merged_model* which is and instance of the class Model, *key* which is a list that contains 'gene_symbol' and 'disease_name' that are the labels where we look for associations, *int* that is an integer number (10).

  This method returns a list with the ten top most frequent associations.

Class GeneDataset

GeneDataset	Model
• Getting labels of the gene dataset • Record the 10 most frequent distinct associations between genes and diseases • Provide the list of sentences correlated to a specific gene, given a gene symbol or its Id • Provide the list of disease (disease name) to which a certain gene is associated	• - • - • - • Model

- Def __init__():

  This init method simply calls the constructor of the base class Model and passes the file path to it. It has been created on the assumption that the Model will not receive a pandas DataFrame, hence the "*is_dataframe*" flag remains False.

- Def get_unique_entries():

  This method is inherited from the class Model. The difference is that we set the argument *key = 'gene_symbol'*, so it returns the sorted Numpy Array that includes all the gene symbols without repetitions.

  The process we apply is called overriding as we provide a specific implementation of the method.

- Def most_frequent_association():

  This method is inherited from the class Model and returns the ten top most frequent associations between the two DataFrames "genes" and "diseases".

- Def get_sentences():

  This method is inherited from the class Model and allows the user to get the list of sentences correlated with the inserted label*(gene_symbol or geneID)* and the inserted attribute.

- Def find_association():

  This method is inherited from the class Model. It works with the merged DataFrame and what changes is that we implement it setting the parameter *second_key = 'disease_name'.* In this way, after having chosen the *key(gene_symbol or geneID)* and the specific attribute, this function return set with all the elements of the *'disease_name'* column that are associated with the elements of the first *key*.

DiseaseDataset

DiseaseDataset	Model
• Getting labels of the disease dataset • Record the 10 most frequent distinct associations between diseases and genes • Provide the list of sentences correlated to a specific disease, given a disease name or its Id • Provide the list of genes (gene symbol) to which a certain disease is associated	• - • - • - • Model

- Def __init__():

  This init method simply calls the constructor of the base class Model and passes the file path to it. It has been created on the assumption that the Model will not receive a pandas DataFrame, hence the "*is_dataframe*" flag remains False.

- Def get_unique_entries():

  This method is inherited from the class Model. The difference is that we set the argument *key = 'disease_name'*, so it returns the sorted Numpy Array that includes all the disease names without repetitions.

  The process we apply is called overriding as we provide a specific implementation of the method.

- Def most_frequent_association():

  This method is inherited from the class Model and returns the ten top most frequent associations between the two DataFrames “genes” and “diseases”.

- Def get_sentences():

  This method is inherited from the class Model and allows the user to get the list of sentences correlated with the inserted label*(disease_name or diseaseID)* and the inserted attribute.

- Def find_association():

  This method is inherited from the class Model. It works with the merged DataFrame and what changes is that we implement it setting the parameter *second_key = 'gene_symbol'.* In this way, after having chosen the *key(disease_name or diseaseID)* and the specific attribute, this function return set with all the elements of the *'gene_symbol'* column that are associated with the elements of the first *key*.

Controller.py

The Controller module has the role of coordinating both the web app and the Gene and Disease models.
An initial check is performed to use the correct path format depending on the machine OS on which the app is running (Windows uses two double forwards slash while unix based OS use one back slash). After that the flask web app is created and the different routes are created.
The three routes used by the app are:

- "/": landing page.
- "/data/<datatype>": which depending on the datatype requested (i.e. Gene model or Disease model) provides the information of that model and the two input based operations that can be performed on the dataset, informations relevant for the rendering of the page are passed to the render_template method.
- "/results/<operation>": this page is used to provide the result to the query the user inputs in the "/data/" page, the results of the query are passed to the page (the methods of the Model class return a string to signal that no results were found or that something was wrong with the query in case some error happens).

Programming project: HTML templates and CSS files

The layout of all pages was created using **Bootstrap 5**, a CSS framework often used for its ability to render sites responsive (i.e. easily usable both on mobile and pcs).
In our case bootstrap was mainly used for its grid system, which easily allows the placement of html elements using its container and rows inside a predefined flexbox.
Moreover Bootstrap offers a wide variety of styles to apply to html elements such as tables and lists, which we used to display data in a more captivating way.
For some elements the use of CSS was necessary such as setting a background for each page but also to create a scrollable list that didn't overflow the entire page, by fixing its height (this element is present when visualizing the Gene and Disease models); additionally some minor changes for each page are present in each CSS file to make up for the color of the background (for example changing the color of the footer to make it more readable).
Two CSS files were created to apply different styles to each page (genes.css and diseases.css) and the values that were passed to the render_template method were then used on the pages by taking advantage of the Jinja2 templating engine present in Flask.
The home page ("home.html") is used as a landing page and it has a general description of the DisGeNET database and of the goal of the site.
The data handling page ("data.html") is used to visualize data and to input query.
Finally the results page ("results.html") is used to visualize the results of a given query inputed in the data.html page: in the case of queries that require as an output one or more sentences the usage of the **|safe** keyword is used to tell the Jinja2 engine to render the html present in the sentences.

N.B: **The insights of specific part of the HTML code are described as comment in the HTML files. While the description of methods of the classes that we created are well explained inside this document.**