Amelia Kurti, Ilaria Misuri, Pietro Zafferani

Software Application ‘s project documentation file

The project is about creation of a dataset reader that can perform operations on them. It is thought as an interactive user-friendly interface, allowing the selection of desired operations and, when needed, parameters for them. The project relies on three components:

1. User-interface: web pages allowing communication between the user and the underlying software
2. Bridging software: sort of API that can pass inputs from the user-interface to the core part, alias the DataFrames-operating program
3. Core: software able to perform operations directly on the DataFrames and pass the outputs through the bridge to the user interface for displaying

USER-INTERFACE

Composed of HTML templates that will directly communicate with the user, displaying operations and recording inputs from the user and pass them to the Flask object. It redirects to the Bridging software and through it the request reaches eventually the Core, thus the computation is performed taking into account the input given by the user if it is needed.

The Flask object is composed of two specific Operations’ instances from the Bridging part (data_g for Gene dataset and data_d for Disease dataset) that will be used in the 14 routes, as following:

- ‘`/`’ → Homepage: presents all the possible operations in form of links together with a summary description.
- ‘`/shapes`’ → Shapes: presents the dimensions of both datasets.
- ‘`/semantics`’ → Semantics: presents the each column’s label of both datasets.
- ‘`/list.genes`’→ List_genes: presents all the genes listed according their abundancy.
- ‘`/list.diseases`’ → List_disease: presents all the diseases listed according their abundancy.
- ‘`/sentences.Gene`’→ g_sentences: presents the choice for the user whether to search for genes filtering by ID or by symbol. The input will be recorded and passed to the specific route by the method ‘POST’ and the form ‘action’. Moreover a drop-down button contains suggestions that might be useful for the user.
- ‘`/sentences.Gene.result`’ → gs_results: presents a list of all the sentences that correlate COVID to the gene of interest. If no association is found the webpage will display a message as well as if errors occurs in user’s input. The HTML file will be provided with specific values for variables according to the user’s demands (i.e CHOICE).
- ‘`/sentences.Disease`’ → d_sentences: presents the choice for the user if to search for disease filtering by ID or by name. The input will be recorded and passed to the specific route by the method ‘POST’ and the form ‘action’. Moreover a drop-down button contains suggestions that might be useful for the user.
- ‘`/sentences.Disease.result`’ → ds_results: presents a list of all the sentences that correlate COVID to the disease of interest. If no association is found the webpage will display a message as well as if errors occur in user’s input. The HTML file will be provided with specific values for variables according to the user’s demands (i.e. CHOICE).
- ‘`/top10`’ → top_10: presents a list of the 10 most frequent Gene-Disease associations.
- ‘`/associations.gene`’ → g_association: presents the choice for the user whether to select gene filtering by ID or by symbol. The input will be recorded and passed to the specific route by the method ‘POST’ and the form ‘action’. Moreover a drop-down button contains suggestions that might be useful for the user.
- ‘`/associations.Gene.result`’ → g_ass_result: presents the list of all the diseases associated with the selected gene. The HTML file will be provided of specific values for variables according to the user’s demand (i.e. CHOICE).

- '`/associations.disease`' → d_association: presents the choice for the user if to select gene filtering by ID or by name. The input will be recorded and passed to the specific route by the method 'POST' and the form 'action'. Moreover a drop-down button contains suggestions that might be useful for the user.
- '`/associations.Disease.result`' → d_ass_result: presents the list of all the diseases associated with the selected gene. The HTML file will be provided of specific values for variables according to the user's demand (i.e. CHOICE).

All the templates contain a link that redirects to the homepage, instead the templates that require the user's input contain also a link that allows to retry directly the same operation. Some templates are used to display the results of the same operation applied on different datasets, to do so Jinja2 was implemented. This leads to reusability of the code.

BRIDGING SOFTWARE

It is the reader of the datasets, the communication point between the different parts and acts as the mediator of the input/output exchanging flow. It is composed of two classes:

- **Read** class, to have access to the datasets and convert them into Pandas' DataFrames objects. The reading is private so that the user cannot manipulate them ensuring the proper execution of the software.
- **Operations** class, to pass the DataFrames and to coordinate the inputs from the User's interface to the Core and vice versa. The class' attributes are:
  a) __label: the specific name of the DataFrame that defines the object. It determines the main DataFrame used for the object, it is set private so that the user can not improperly access it and affect the performance of whole application.
  b) Filter: is the number of the column used for searching or grouping elements. Set by default to zero.
  c) my_key: is the number that identify the specific element to look for according to the desires of the user.

The **Read** class provides few methods:

a) read_it_g: is a public method that allows accessing to the private method __read_it_g, whose only function is to convert specifically the Gene dataset into a pandas DataFrame. This is essential for every requested operation to be performed.
b) Read_it_d: is a public method that allows accessing to the private method __read_it_d, whose only function is to convert specifically the Disease dataset into a pandas DataFrame. This is essential for every requested operation to be performed.

The **Operations** class has multiple methods:

a) __set_filter: change the filter of the Operations' object. The private method is accessed by means of set_it method because any changes in the selection of the filter are critical for the performance. They are provided to the user with specificity for the selected route and searching filter, so it's directly through the User Interface that the parameter is set properly.
b) set_key: set the searching key of the object. Provided by the user's choice in the specific route, it is the element of interest in the DataFrame under analysis.
c) read_id_gene and read_it_disease: give access to the DataFrames elaborated by the Read class.
d) D_set and D_other: select the right datasets and assign relative functions. The 'set' can be seen as the main DataFrame of interest while the 'other' is the complementary (e.g. 'set' is the Gene

dataset so automatically 'other' is the Diseases' one). They are chosen according to the label of the object created and then properly used for communication with the Core part.

e) __oper_no and __oper_yes: to perform operation-calling. The two private methods are accessed respectively by oper_No and oper_Yes methods, that protect by potentially risky user's manipulations. The methods provide two dictionaries called registry_n and registr_y made of 'OperationName:Core'sOperationInvocation' key:value pairs. The key is directly called by the User interface part (provided as the argument of the functions) and the element itself is an invocation for MetaData object specific operation, performed onto 'set' and 'other' DataFrames. Specific attributes are passed if needed. The bridging is made by means of dictionaries considering the high speed of accessing of the object type. There exist two dictionaries split according to the needing for a user's input or not, to speed up the whole process, natively time-consuming for the size of the datasets. The methods will return the element chosen and the user interface part will display the result.

CORE

It is the software's part that carries out all the analytical operations, it operates over DataFrames thanks to Pandas facilities. It is composed by a unique class called MetaData that is agnostic of the specific DataFrame content. This allows flexibility of the code, meaning that one can use the same method with the same parameters to operate on both datasets.

MetaData class's attributes:

a) __data: is set private and it represent the main DataFrame on which the operations are performed
b) __array: is set private, it is not passed as a parameter but it is automatically created for every instance of MetaData. It is the Numpy array derived from the DataFrame because this type of object speeds up the computations and considering the size of the datasets this implementation modality was a necessity.

The **MetaData** class provides all the methods for solving each specific operation:

a) dimensions: returns a tuple containing the dimensions of the DataFrame attribute, this operation is achieved by invoking the Pandas attribute .shape.
b) semantics: returns a list containing the labels of the DataFrame columns, this operation is achieved by the invocation of the Pandas attribute .columns.
c) list_names: creates a Series containing in ascending order the genes' symbols or the diseases' names according to the type of object passed, eventually it returns a list of the indexes.
d) __all_sentences: is protected since it is designed as an auxiliary function, thus it can be invoked only inside the class scope. It takes as parameters the index of the column (n_column), that contains the (element) parameter, and (item_index) that is instead the index of the column in which the item corresponding to (element) must be searched in, it is set as 1 by default so that it can be directly implemented by the .sentence() method to detect the items in the 'sentence' column. If the parameter (element) is present in the column of the DataFrame with index [n_column] then the item on the same row belonging to the column with index equal to the parameter (item_index) will be added to a list, this allows the programmer to re-use this method for different purposes by changing only one parameter. The possibility of unwelcomed outcomes are taken into account and managed.
e) covid_sentence: takes the same parameters of the previous method except for (item_index) which is set by default. It invokes .__all_sentences() in order to obtain the complete list of sentences related to the (key), if the item which is a HTML sentence is directly related to COVID19 then it will be added to the new list created. Markup() is a module that allows to implement Markup

languages. This method takes into account also the possibility that no correlation with COVID19 is found.

f) __merging: is an auxiliary method that cannot be invoked out of the internal scope of the class, it takes as parameters an instance (other) of the MetaData class and (number) which defines the criterium to which merge the 2 DataFrames with. It returns a new DataFrame created by the built-in Pandas method pandas.merge(), thus it will be used by other methods to solve more complex analytical operations.

g) association: takes as parameters (other) which is the MetaData instance to merge the DataFrame with and (element) which is the item to whom the association is found. The protected method .__merging() is used to create the merged DataFrame in which perform the next operation: number 3 as merging criterium stands for the 'pmid' label because this enables us to find the publications that connects a certain gene to a certain disease and viceversa. The .__all_sentences() method is invoked on the merged DataFrame and it returns a list containing all the items associated to the (element) parameter which will be the only given by the user, the (item_index) parameter is fixed as 9 because it stands for the 'gene_symbol' or 'disease_name' according to the DataFrame used to merge the instance's one with. Eventually a list is returned with all the gene symbols or disease names that are associated to the element passed as parameter.

h) most_frequent: The only parameters taken are those needed by the protected method .__merging(), in particular (number) was set to 3 since it refers to the label of the ['pmid'] column that is the merging criterium, the method uses the pandas built-in function .groupby() to align the element-couples belonging to the columns ['gene_symbol'] and ['disease_name'] sharing the same['pmid']. Since the task requires to return the top 10 discrete associations, we exploit the built-in functions of Pandas .count() and .sort_values() to obtain a descending DataFrame. In the end the method returns a list containing the 10 most frequent discrete associations between and diseases, moreover it is irrelevant which DataFrame is chosen as self and which one as (other) because the output will be the same.