CPS Project A.A. 2020-2021

Andrea Graziani (0273395)
andrea.graziani93@outlook.it
Università degli Studi di Roma "Tor Vergata"
Rome, Italy

1 INTRODUCTION

In this report we will describe main data structures, classes and variables of our application, which can be used to manage several kind of **time series** data, plotting them and determining several useful informations according to our project's specification.

Our application requires both Python `3.x`[1] and *Classic Jupyter Notebook*[2] to run properly, while, in order to perform the front-end rendering, `ipywidgets`[3], a python package containing several interactive HTML widgets for jupyter notebooks, is required. Anyhow, make sure you to satisfy all dependencies listed inside the `README` file before running this application on your system.

We remind that the source code of this application is available on `GitHub`[4], including the LaTeX source code of this report.

2 DESCRIPTION

2.1 Jupyter Notebook Description

In order to run the application, just run one of the *two* code cell inside `CPS.ipynb` jupyter notebook file:

(1) Running the first one, the user interface reported in **fig. 1 (a)** will be shown and, using the interactive widgets displayed, you can plot a selected dataset applying various customizations.
From an implementation point of view, this user interference is provided by `GUIPlot` class which exports following methods:
   (a) `__build` method which includes all necessary code to instantiate all widgets including the so-called geometry management (that is where to put those widgets).
   (b) `display` method which, instead, is used to clean current jupyter notebook code cell and show all previously built widgets as output.

(2) Conversely, running the second code cell, will be displayed the user interface reported in **fig. 1 (b)**, using which you can compare regression lines.
This interface is built by `GUIRegressionLineComparison` class which operates identically to `GUIPlot` class, despite it builds and manages different widgets.

To be precise, all tasks related to *event handling*, *widget callbacks function* and *screen update* is managed by both `GUIPlotController` and `GUIRegressionLineComparisonController` classes; however, we preferred to omit a detailed description of all classes involved into GUI rendering because they are just too technical and so not relevant for the purposes of this report.

[1] https://www.python.org/
[2] https://jupyter.org/
[3] https://ipywidgets.readthedocs.io/en/latest/
[4] https://github.com/AndreaG93/CPS-Project

2.2 `Application` Class Description

However, to display any user interface, an instance of `Application` class is required.

From an architectural point of view, the `Application` class is used to model the internal state of our application, holding all references to available datasets and plot customization options. To be more precise, this class has following responsibilities:

- To instantiate and to maintain a reference to all `TimeSeriesDataset` objects which, as explained later, are used to model all datasets.
  This task can be done running `__build_dataset_registry` method which firstly reads all `csv` files stored inside `/data` directory, then, after reading the header of datasets, instantiate the correct subclass of `TimeSeriesDataset` class according to available columns inside the dataset itself. Do not worry, we will explain the characteristics of `TimeSeriesDataset` class and of its subclasses later.
  All instantiated `TimeSeriesDataset` object are finally stored as *key-value* pair inside the dictionary `self.__dataset_registry`, where the *key* is the file-name of corresponding dataset.
- To instantiate and to hold references to `ApplicationOptions` objects, which are used, instead, to store plots customization options, like *time-range* or *month filter*, specified by the user using the user interface.
  This activity is performed running `__build_options_registry`, which builds a `ApplicationOptions` object for each `TimeSeriesDataset` object stored inside `self.__dataset_registry` field.
  Similarly to the previous case, `ApplicationOptions` objects are stored into a dictionary data structure, using dataset file-name as keys.

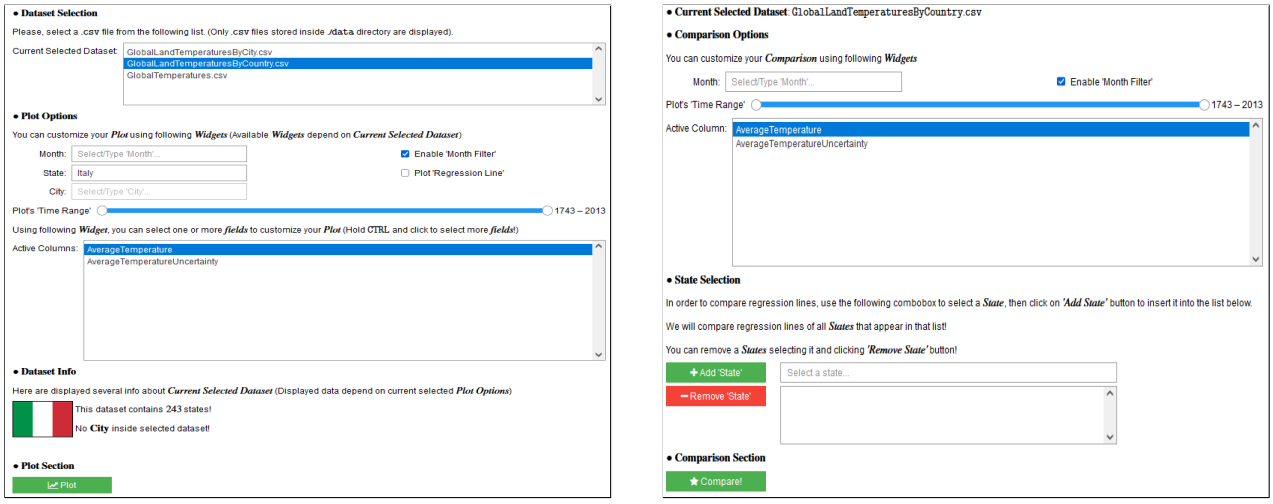
2.3 Datasets

According to project's specification, we have used the datasets belonging to *Global Climate Change Data*[5], which include **time series data** where each instance, indexed by a *time-stamp* represented by a string using the `ISO 8601` format `YYYY-MM-DD` without seconds, contains several observations about temperatures measured in several cities and states around the world.

Therefore, we have modelled aforementioned datasets using `TimeSeriesDataset` class, whose responsibilities are:

(1) Read data from file system running `__read_data` method, returning a `pandas.DataFrame` object whose index is represented by `DatetimeIndex` object, obtained parsing time-stamps of the `dt` column inside the dataset file exploiting the `pandas.read_csv` function.

[5] You can download the dataset from: https://data.world/data-society/global-climate-change-data



(a) (b)

Figure 1: Application's GUI.

(2) Determine the time range of available data, which are needed to build the user interface, running `__compute_time_range`.
(3) Determine which columns contain numeric data running `__compute_numeric_type_columns`.

2.4 Datasets Files

Respecting project's specifications, our application is capable to manage following datasets files:

`GlobalLandTemperaturesByCity.csv` This dataset includes the observations of *Average Temperature* and *Average Temperature Uncertainty*, collected from *1743-11-01* to *2013-09-01*, regarding several *states* and *cities* of the world.
From an implementation point of view, this dataset is represented by `TimeSeriesDatasetGlobalClimateChange` class, a subclass of `TimeSeriesDataset`.
This class exports several methods used to manage all data regarding states and cities of all available observations, reading data from `country` and `city` columns inside the dataset file.
For example, `get_state_list` and `get_city_list` methods are used to compute a list of all available states and cities respectively inside the dataset.
Conversely, `get_city_list_belonging_to_state` method is used to determine all cities belonging to a specified state.
Moreover, this class exports the `get_state_of_city` method, which can be used to know the state to which a specified city belongs.
Please note that, in order to save computing resources by avoiding unneeded computations, all informations about cities and states are computed exploiting *lazy evaluation*, that is aforementioned data are evaluated only on the first need and stored them in case they would be needed again later.

`GlobalLandTemperaturesByCountry.csv` This dataset is similar to the previous one except for the absence of `city` column, therefore `TimeSeriesDatasetGlobalClimateChangeNoCity` class was used to model this dataset. Observe that it is a subclass of `TimeSeriesDatasetGlobalClimateChange` and, as you can see from code, to reflect the absence of data about cities, we have performed method overriding accordingly.

`GlobalLandTemperatures.csv` This dataset includes data from *1750-01-01* to *2015-12-01* regarding several kinds of observations like *Land Average Temperature*, *Land Average Temperature Uncertainty* and so on.
Since this dataset file contains neither `country` and `city` columns, `TimeSeriesDatasetGlobalClimateChangeNoStateNoCity` class was been used and, like previously, we have performed method overriding accordingly to the characteristics of this dataset.

2.5 The *Datasets Info* and *Plot Options* sections

All aforementioned methods are used to properly manage the user interface, setting and updating the contents of all widgets inside the "*Plot Options*" section of the GUI (see **fig. 1 (a)**); for example, `get_state_list` and `get_city_list` methods are used to populate available choices of both "*State*" and "*City*" combo-box widgets.

However, `TimeSeriesDatasetGlobalClimateChange` class methods allow us to determine several informations about a selected dataset too.

In fact, application's user interface provides another section, called "*Dataset Info*", where following information are displayed to the user when a dataset, equipped with both `country` and `city` columns, is selected:

(1) The total number of states or nations represented inside current selected dataset.
(2) The total number of cities available inside the entire current selected dataset.

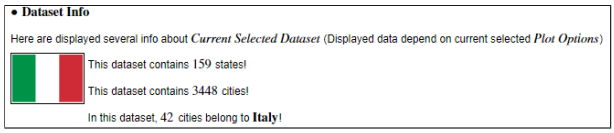


Figure 2: An example of *Dataset info* section output

Moreover, according to project's specification, after selecting a state will be displayed the number of cities belonging to it.

For example, if you select `GlobalLandTemperaturesByCity.csv` dataset specifying *Italy* as selected state, the output, reported in **fig. 2**, will be shown.

Furthermore, using `get_city_list_belonging_to_state` method, is very simple to know which cities belongs to a selected state and select it in order to customize your plot. **fig. 3** shows the output displayed when *Italy* state is set and *City* combo-box is selected.

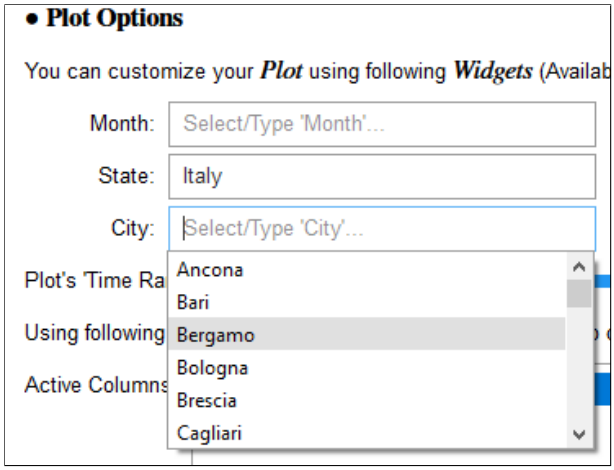


Figure 3: The City Combo-box displaying only Italian cities.

The flag of the selected state, displayed inside "*Datasets Info*" section, is retrieved exploiting `RESTful API` belonging to a provider called `REST COUNTRIES`[6].

From an implementation point of view, all requests are forwarded using `__get_url_from_rest_api` static method, implemented inside `StateFlagUrlRegistry` class whose structure is based on so-called *registry design pattern*, which was been adopted to simplify the storing and the retrieving of state flags without forwarding unneeded requests.

2.6 Plot

When you will finish to set all needed plot customization options, you can plot your data clicking the button "*Plot*" displayed inside "*Plot Section*". The click event triggers the invocation of `plot_event` method which manages all tasks necessary for plotting data.

The most important work is done by `get_filtered_data` method exported by `TimeSeriesDatasetGlobalClimateChange` class, which includes all application logic used to perform following tasks:

(1) apply plot customization options to selected data, which is done exploiting `pandas` python package.
(2) check correctness of user input.
(3) pre-process data, looking for missing records and `NaN` values. This activity is done exploiting `pandas.DataFrame.reindex`[7] method, according to which the `DataFrame` object is conformed to new index built using `pandas.date_range`[8] method using a *sampling frequency* (the number of sampling points per unit time) equal to that of the dataset, which is equal at one data point every first day of every month.
In this way, we are able to find missing records which, according to our experiments, are present inside every dataset[9].
Finally, to manage `NaN` values, we have simply exploited `interpolate`[10] and `dropna`[11] functions.

Finally, invoking `plot` function, the plot will be displayed like in **fig. 4**, inside which a regression line is displayed too.

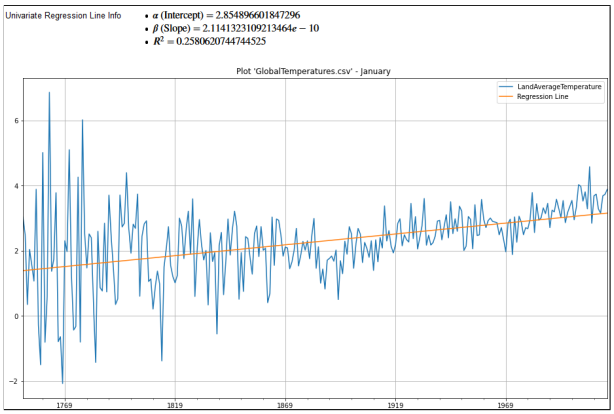


Figure 4: Output plot

3 REGRESSION LINE COMPUTATION

In this last section, we will describe how a regression line is computed.

3.1 `SimpleRandomSample` Class Description

The `SimpleRandomSample` class is used to model a **random sample** holding the values $x_1, x_2, ..., x_n$ of a random sample $X_1, X_2, ..., X_n$ of size $n \in \mathbb{N}$, where $X_1, X_2, ..., X_n$ are *independent random variables* that satisfy following equation:

$$X \overset{d}{=} X \qquad \forall k = 1, ..., n \tag{1}$$

that is, $X_1, X_2, ..., X_n$ converge in distribution to a real random variable X.

[6] https://restcountries.eu/

[7] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reindex.html

[8] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.date_range.html

[9] Please note that the number of missing records is suppressed by default; however, you can print it de-commenting some line of code inside `get_filtered_data` method.

[10] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.interpolate.html

[11] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

This class allows us to compute following *statistics* of a given random sample:

Sample Mean which is computed by `get_mean` function using following formula:

$$\overline{x}_n \overset{def}{=} \frac{1}{n}\sum_{k=1}^{n} x_k \tag{2}$$

Unbiased Sample Variance which is determined exploiting `get_unbiased_variance` function using following equation:

$$s_n^2(X) \overset{def}{=} \frac{1}{n-1}\sum_{k=1}^{n}(x_k - \overline{x}_n)^2 \tag{3}$$

Sample Covariance which is, instead, provided by a static method, called `calc_sample_covariance`, using following equation:

$$s_n(X,Y) \overset{def}{=} \frac{1}{n-1}\sum_{k=1}^{n}(x_k - \overline{x}_n)(y_k - \overline{y}_n) \tag{4}$$

3.2 `UnivariateRegressionLine` Class Description

All statistics computed by a `SimpleRandomSample` instance are used to compute regression lines, which are been modelled using `UnivariateRegressionLine` class.

To be more precise, that class holds all data needed to represent a *linear regression function* $f : \mathbb{R} \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ given by:

$$f(x;\theta) \overset{def}{=} \alpha + \beta x \qquad \forall x \in \mathbb{R} \tag{5}$$

where $\theta \equiv (\alpha, \beta) \in \mathbb{R}^2$ is the *regression parameter*, where α is called *intercept* while β is knowns as *slope*.

Regression parameter is computed by `__calc_slope_and_intercept` method, using following equations:

$$\alpha = \overline{y}_n - \beta\overline{x}_n \qquad \beta = \frac{s_n(X,Y)}{s_n^2(X)} \tag{6}$$

Then, all points of the regression line $(x_k, \hat{y}_k)_{k=1}^n$ are computed by `__calc_fitted_values` method using following equation:

$$\hat{y}_k \overset{def}{=} \alpha + \beta x_k \qquad \forall k = 1, 2, ..., n \tag{7}$$

From an implementation point of view, the regression line is computed running `compute_univariate_regression_line` function.

However, `UnivariateRegressionLine` class has another very important responsibility: to perform regression lines comparison tasks.

Regression lines comparison is needed to determine which state, belonging to a previously user defined list, has the *worst* regression line in a specified time range. To perform this task, we exploited the *coefficient of determination*, also denoted R^2, which is the positive number defined as follows:

$$R^2 \overset{def}{=} \frac{ESS}{TSS} \tag{8}$$

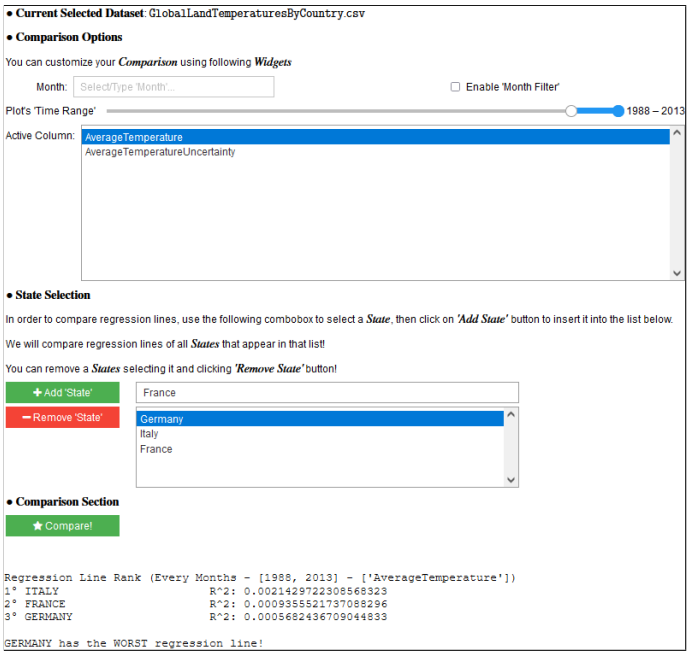


Figure 5: Which country between *Italy*, *France* and *Germany* has the *worst* regression line in the last 25 years?

where *ESS* is the so-called *explained sum of squares* while *TSS* is the *total sum of squares*, which are computed by `__calc_ess` and `__calc_tss` respectively, using following equations:

$$ESS \overset{def}{=} \sum_{k=1}^{n}(\hat{y}_k - \overline{y}_n)^2 \qquad TSS \overset{def}{=} \sum_{k=1}^{n}(y_k - \overline{y}_n)^2 \tag{9}$$

The R^2 value is very useful because tell us how much the fitted/predicted data $\hat{y}$, explains the actual data y. In particular, the higher the value of R^2 the more successfully the variation in the data set $(y_k)_{n=1}^k$ can be explained in terms of the linear dependence on the data set $(x_k)_{n=1}^k$ expressed by the regression line. The R^2 value ranges from 0 to 1, with higher values denoting a strong fit, and lower values denoting a weak fit.

To perform this task, we exploited `rank_regression_lines` static method defined inside `UnivariateRegressionLine` class. After the comparison of regression lines, as you can see from the example reported in **fig. 5**, will be finally displayed as output a rank of regression lines based on best R^2 values printing which state has the worst regression line.