timeSeries-processing

Library which processes time series datasets

Installation

```
pip install timeSeries-processing
```

References

- NumPy, the fundamental package for scientific computing with Python.
- Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- scikit-learn, machine Learning in Python.

License

MIT

DESCRIPTION

timeSeries-processing is a python library which processes time series datasets.
The supported time series are daily time series, which means that the dates are always days.

Purpose

This library is a tool for time series modeling. In particular, it is an auxiliary utility for helping building machine learning models for time series forecasting.

In fact, the main application of this library is to, given a time series dataset, add some useful and interesting time-related features to it.
In other words, it allows the user to extract and build some important time-related explanatory features.

These new features are obtained from a specific and already existing feature of the dataset, by selecting, grouping and processing the days which are somehow related to the ones in the given dataset.
As a result, each of these new computed features is an indicator of the behaviour of the specific feature but in other related days.

For example, given a time series dataset and specifying a certain feature, it is possible to add some new features representing the specified feature but in the previous days.
Each new feature indicates the value of the specified feature in a certain previous day.

The interfaces of the functionalities of the library are simple and intuitive, but they are also rich. In this way, the user is able to personalize the time series operations in a powerful and flexible way.

Functionalities

There are three groups of functionalities.

The first group is able to manipulate dates (i.e. days). There are several different operations. For example, one of them is able to split a collection of days by a certain criterion, which can either be year, month or season.
These functionalities are mainly built in order to be some auxiliary utilities for the other functionalities.

The second group is able to plot time series values.
The user can specify several different options, in order to change the visualization and the division of the values. This can be particularly useful for understanding some time-related patterns, like seasonal behaviours.

The third group of functionalities is the most important. These are the processing functionalities, i.e. the ones which actually process the time series datasets.
As described above, the main purpose of these functionalities is to extract and build interesting time-related explanatory features.

Implementation details

This library is built on top of the pandas library.
The pandas built-in data types are indeed used.

- The dates are represented with the `pd.Timestamp` type.
- Vectors of dates are represented with the `pd.DatetimeIndex` type.
- The time series datasets are represented as `pd.DataFrame` indexed by dates (i.e. the index is a `pd.DatetimeIndex`).

In addition, several pandas utilities and methods are used.

Each processing functionality of timeSeries-processing adds the new extracted features to the given dataset by producing a new dataset, i.e. the given dataset is not modified.
In addition, each processing functionality also returns two NumPy arrays: the first is `X`, which contains the explanatory features of the returned dataset; the second is `y`, which contains the response feature of the returned dataset.
In other words, each of these functionalities automatically splits the obtained dataset into the features used to make the predictions and the feature which is the target of the prediction.
This can be particularly useful to easily build and evaluate different machine learning models in a compact way.

To conclude, the time series plotting is built on top of the Matplotlib library.

DOCUMENTATION

Functions to manipulate dates

```
def find_missing_days(days)
```

Return, given a vector of days, his missing days.

More specifically, the missing days are the ones which are not present in the contiguous sequence of days in `days`.

Parameters:

- `days: pd.DatetimeIndex`
  Vector of dates.

Returns: `pd.DatetimeIndex`
Vector of missing days.

```
def find_same_month_days(day)
```

Return, given a day, all the days which are in the same month.

Parameters:

- `day: pd.Timestamp`

Returns: `pd.DatetimeIndex`
Vector of the days in the same month.

```
def find_same_season_days(day)
```

Return, given a day, all the days which are in the same season.

The meteorological seasons are considered, and not the astronomical ones.

Parameters:

- `day: pd.Timestamp`

Returns: `pd.DatetimeIndex`
Vector of the days in the same season.

```
def find_k_years_ago_days(day, k=1, n_days=11)
```

Return, given a day, the days which are centered on that day but k years ago.

Parameters:

- `day: pd.Timestamp`
- `k: int`
  Indicates which past year has to be considered (i.e. k years ago).
- `n_days: int` or `str`
  Indicates specifically which are the `k` years ago to select.
  If it's an `int`, it must be an odd positive number. The `n_days` centered on `day` but `k` years ago are selected.
  If it's a `str`, it must be either `"month"` or `"season"`. All the days in the same month/season but `k` years ago are selected. (The meteorological seasons are considered, and not the astronomical ones)

Returns: `pd.DatetimeIndex`
Vector of the selected days.

Raises: `ValueError`
When `n_days` is neither an odd positive integer nor `"month"` nor `"season"`.

```
def find_current_year_days(day, n_days=11, current_day=False)
```

Return, given a day, the preceding days of the same year which are centered on that day.

Parameters:

- `day: pd.Timestamp`
- `n_days: int` or `str`
  Indicates specifically which are the current year days to select.
  If it's an `int`, the `n_days` preceding `day` are selected.
  If it's a str, it must be either `"month"` or `"season"`. All the days in the same month/season that precede `day` are selected. (The meteorological seasons are considered, and not the astronomical ones)
- `current_day: bool`
  Indicates whether to select also the current day (i.e. `day`) or not.

Returns: `pd.DatetimeIndex`
Vector of the selected days.

Raises: `ValueError`
When `n_days` is neither an integer nor `"month"` nor `"season"`.

```
def group_days_by(days, criterion)
```

Group the given vector of days according to the given criterion.

Parameters:
- `days: pd.DatetimeIndex`
- `criterion: str`
  Indicates how to group the given days. It can be either `"year"` or `"month"` or `"season"`. (The meteorological seasons are considered, and not the astronomical ones)

Returns: `list`
List of pairs (i.e. tuples). Each pair is a group of days.
- The first element is a string which represents the group name (i.e. group label).
- The second element is the vector of days in that group, i.e. it's a `pd.DatetimeIndex`.

Raises: `ValueError`
When `criterion` is neither `"year"` nor `"month"` nor `"season"`.

-[Notes]-
For the sake of completeness, it's important to say that if `criterion` is either `"month"` or `"season"`, also days of different years could be grouped together.

Function to plot a time series

```
def plot_timeSeries(df, col_name, divide=None, xlabel="Days",
line=True, title="Time series values", figsize=(9,9))
```

Plot a column of the given time series DataFrame.

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- `col_name: str`
  Indicates the specified column to plot.
- `divide: str`
  Indicates if and how to divide the plotted values. It can either be None, `"year"`, `"month"` or `"season"`. (The meteorological seasons are considered, and not the astronomical ones). That division is simply made graphically using different colors.
- `xlabel: str`
  Label to put on the x axis.
- `line: bool`
  Indicates whether to connect the points with a line.
- `title: str`
  Title of the plot.
- `figsize: tuple`
  Dimensions of the plot.

Returns: `matplotlib.axes.Axes`
The matplotlib Axes where the plot has been made.

Processing functions

```
def split_X_y(df, y_col=None, scale_y=True):
```

Split the given DataFrame into X and y.

X is a matrix which contains the explanatory variables of `df`, y is a vector which contains the response variable of `df` (i.e. the variable which is the target of the prediction analysis tasks). Optionally, the values in y can be scaled.

This function is an auxiliary utility for the processing functions.

Parameters:

- `df: pd.DataFrame`
- `y_col: str`
  Indicates which is the `df` column that is the response feature.
  If it is None, the last `df` column is considered.
- `scale_y: bool`
  Indicates whether to scale or not the values in y.

Returns:

- `X: np.array`
  Two-dimensional np.array, containing the explanatory features of `df`.
- `y: np.array`
  Mono dimensional np.array, containing the response feature of `df`.

-[Notes]-
The scaling of the values in y is performed using the sklearn MinMaxScaler.

```
def add_timeSeries_dataframe(df, df_other, y_col=None,
scale_y=True)
```

Add to a time series DataFrame another time series DataFrame.

The two DataFrames are concatenated into a new DataFrame, i.e. the resulting DataFrame contains all the columns in `df` and `df_other`. This concatenation is done with respect to the former DataFrame: this means that all the days of `df` are kept, while only the days of `df_other` that are also in `df` are kept.

In addition, the resulting DataFrame is automatically split into the X matrix and the y vector, which are respectively the matrix containing the explanatory features and the vector containing the response feature. (The response feature is the one which is the target of the prediction analysis tasks).

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- `df_other: pd.DataFrame`
  Other DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`), which has to be added to the former.
- `y_col: str`
  Indicates which is the column of the resulting DataFrame to be used as y column.
- `scale_y: bool`
  Indicates whether to scale or not the values of the response feature y.

Returns:

- `pd.DataFrame`
  The DataFrame resulting from the concatenation.
- `X: np.array`
  Two-dimensional numpy array which contains the explanatory features.
- `y: np.array`
  Mono-dimensional numpy array which contains the response feature.

```
def add_k_previous_days(df, col_name, k, y_col=None, scale_y=True)
```

Add, to a time series DataFrame, features containing values of the specified column but related to the previous days.

A new DataFrame is built, which is created from `df` adding k new columns. These k new columns contain the values of the column `col_name` but with regard to, respectively: the day before; the two-days before; ... ; the k-days before. In this way, in the resulting DataFrame for each day there is information about the feature `col_name` up to k days before. These k columns are, respectively, called: `"col_name_1"`, `"col_name_2"`, ..., `"col_name_k"`.

The first k days are removed from the resulting DataFrame: that is because for the first k days there isn't enough information to build the new k columns.

In addition, the resulting DataFrame is automatically split into the X matrix and the y vector, which are respectively the matrix containing the explanatory features and the vector containing the response feature. (The response feature is the one which is the target of the prediction analysis tasks).

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- `col_name: str`
  Indicates which is the column to be used to build the k new columns.
- `k: int`
  Indicates how many previous days are to be taken into account (i.e. how many new columns are built).
- `y_col: str`
  Indicates which is the column of the resulting DataFrame to be used as y column.
- `scale_y: bool`
  Indicates whether to scale or not the values of the response feature y.

Returns:

- `pd.DataFrame`
  The DataFrame resulting from the concatenation.
- `X: np.array`
  Two-dimensional numpy array which contains the explanatory features.
- `y: np.array`
  Mono-dimensional numpy array which contains the response feature.

Raises: ValueError
When `df` does not contain a contiguous sequence of days (i.e. there are missing days in `df`).

```
def add_k_years_ago_statistics(df, df_k_years_ago, k=1,
days_to_select=11, stat="mean", columns_to_select=None,
replace_miss=True, y_col=None, scale_y=True)
```

Add, to a time series DataFrame, statistics computed on the other given time series DataFrame, but with respect to the days of k years ago.

`df_k_years_ago` should contain days of k years ago with respect to the days of `df`.

(Nevertheless, both `df` and `df_k_years_ago` can contain multiple years). Let *m* be the number of the selected columns of `df_k_years_ago` (by default all the columns, see the `columns_to_select` parameter). A new DataFrame is built, which is created from `df` adding *m* new columns. (The resulting DataFrame has the same index of `df`). These new *m* columns contain the values computed from the associated columns of `df_k_years_ago` considering the days of *k* years before the ones in `df`.

Going into the details, let *day* be a row of `df`, and *new_column* be one of the *m* new columns created in the resulting DataFrame. The value put in that column for that day is computed from the associated column of `df_k_years_ago` considering the days of `df_k_years_ago` that are centered on *day* but `k` years ago. (See the `find_k_years_ago_days` function). Once the `k` years ago days in `df_k_years_ago` are selected, an unique value for the new column *new_column* and for the day *day* is computed applying a certain statistical aggregation (specified by the input parameter `stat`) on the values of these selected days in the column of `df_k_years_ago` associated to *new_column*.

`days_to_select` specifies, for each *day* of `df`, which `k` years ago days in `df_k_years_ago` are selected. The semantics is quite similar to the parameter `n_days` of the `find_k_years_ago_days` function (it can be either an odd integer or `"month"` or `"season"`).

Actually, `days_to_select` can also be more powerful than that. `days_to_select` can be a predicate (i.e a function that returns a bool), which is used to select the days of `k` years ago: for each *day* of `df`, the `k` years ago days in `df_k_years_ago` for which the function `days_to_select` returns `True` are selected. So, `days_to_select` is a predicate that, in a flexible way, selects the days of `k` years ago . The signature of the function must be:

```
(day: pd.TimeStamp, df: pd.DataFrame,
 day_k_years_ago: pd.TimeStamp,df_k_years_ago: pd.DataFrame):
 bool.
```

Where:

- `day` is the current day of `df` ;
- `df` is the given DataFrame ;
- `day_k_years_ago` is the day of `k` years ago contained in `df_k_years_ago`;
- `df_k_years_ago` is the other given DataFrame, containing days of `k` years ago.

The function returns `True` if and only if `k_years_ago_day` is a day that has to be selected for `day`.

For a certain *day* of `df` it could happen that no `k` years ago day is selected. This means that this *day* has a missing value for each of the *m* new columns (i.e. *m* missing values). In this case, if `replace_miss` is `True`, all the missing values are filled: the missing value for the new column `new_column` is filled computing the mean of all the values in the associated column in `df_k_years_ago`. Otherwise, if `replace_miss` is `False`, the *m* missing values are simply kept as Nan.

So, in the end, *m* new columns are created in the resulting DataFrame, from the selected *m* columns of `df_k_years_ago`. From the selected column with name `"col"` of `df_k_years_ago`, the corresponding column `"k_years_ago_col"` is created in the resulting DataFrame.

In addition, the resulting DataFrame is automatically split into the X matrix and the y vector, which are respectively the matrix containing the explanatory features and the vector containing the response feature. (The response feature is the one which is the target of the prediction analysis tasks).

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- df_k_years_ago: pd.DataFrame
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`). It should contain days of `k` years ago with respect to the days in `df`.
- `k: int`
  Indicates which previous year, with respect to the days in `df`, has to be taken into account (i.e. the year which is `k` years ago). It must be a positive integer.
- `days_to_select: int` or `str` or `callable`
  Indicates, for each day of `df`, which `k` years ago days are selected in `df_k_years_ago`. It must either be an odd integer or `"month"` or `"season"` or a predicate (i.e. a function that returns a boolean). The function signature must be `(day: pd.TimeStamp, df: pd.DataFrame, day_k_years_ago: pd.TimeStamp, df_k_years_ago: pd.DataFrame): bool`
- `stat: str`
  Indicates the statistical aggregation to perform, for each day of `df`, on the selected `k` years ago days of `df_k_years_ago`. It can either be `"mean"` or `"min"` or `"max"`.
- `columns_to_select: list`
  List of strings which indicates the columns of `df_k_years_ago` that have to be taken into account. If it's None, all the columns of `df_k_years_ago` are considered.
- `replace_miss: bool`
  Indicates whether to fill the missing values or keep them as Nan. (The missing values are generated for each day of `df` for which no k years ago day in `df_k_years_ago` is selected).
- `y_col: str`
  Indicates which is the column of the resulting DataFrame to be used as y column.
- `scale_y: bool`
  Indicates whether to scale or not the values of the response feature y.

Returns:

- `pd.DataFrame`
  The resulting DataFrame.
- `X: np.array`
  Two-dimensional numpy array which contains the explanatory features.
- `y: np.array`
  Mono-dimensional numpy array which contains the response feature.

Raises: `ValueError`

- When k is not a positive integer.
- When `stat` is neither `"mean"` nor `"min"` nor `"max"`.

Warns: `UserWarning`

- When, for a day of `df`, no k years ago day in `df_k_years_ago` is selected.
- When, for a day of `df`, less k years ago days are found compared to the ones expected. (This can happen only if `days_to_select` is either an odd integer or `"month"` or `"season"`).

-[Notes]-
If `add_k_years_ago_statistics` is applied multiple times with the same k on the same `df` and `df_k_years_ago`, columns with the same name are potentially created. For instance, if `add_k_years_ago_statistics` is applied three times with the same k on the same DataFrames, from the `df_k_years_ago` column `"col"` three different columns with the same name `"k_years_ago_col"` are potentially created . To avoid that, `add_k_years_ago_statistics` ensures that all the different columns with the same name are properly disambiguated, using progressive numbers. (E.g three different columns with the same name `"k_years_ago_col"` became `"k_years_ago_col"`, `"k_years_ago_col.1"` and `"k_years_ago_col.2"`).

```
def add_current_year_statistics(df, df_current_year,
days_to_select=11, current_day=False, stat="mean",
columns_to_select=None, replace_miss=True, y_col=None,
scale_y=True)
```

Add, to a time series DataFrame, statistics computed on the other given time series DataFrame, with respect to the preceding days of the same year.

`df_current_year` should contain days of the same year with respect to the days of `df`.(Nevertheless, both `df` and `df_current_year` can contain multiple years). Let *m* be

the number of the selected columns of `df_current_year` (by default all the columns, see the
`columns_to_select` parameter). A new DataFrame is built, which is created from `df` adding *m* new columns. (The resulting DataFrame has the same index of `df`). These new *m* columns contain the values computed from the associated columns of `df_current_year` considering the preceding days of the same year with respect to the days in `df`.

Going into the details, let *day* be a row of `df`, and *new_column* be one of the *m* new columns created in the resulting DataFrame. The value put in that column for that day is computed from the associated column of `df_current_year` considering the preceding days of the same year, in `df_current_year`, that are centered in *day*. (See the `find_current_year_days` function). Once the preceding days of the same year are selected from `df_current_year`, an unique value for the new column *new_column* and for the day *day* is computed applying a certain statistical aggregation (specified by the input parameter `stat`) on the values of these selected days in the column of `df_current_year` associated with *new_column*.

`days_to_select` specifies, for each *day* of `df`, which preceding days of the same year are selected from `df_current_year`. The semantics is quite similar to the parameter `n_days` of the `find_current_year_days` function (it can either be an integer or `"month"` or `"season"`).

Actually, `days_to_select` can also be more powerful than that. `days_to_select` can be a predicate (i.e. a function that returns a `bool`), which is used to select the same year days: for each *day* of `df` the preceding same year days of `df_current_year` for which the function `days_to_select` returns `True` are selected. So, `days_to_select` is a predicate that, in a flexible way, selects the same year days. The signature of the function must be:

```
(day: pd.TimeStamp, df: pd.DataFrame,
 day_current_year: pd.TimeStamp,
 df_current_year: pd.DataFrame): bool.
```

Where:

- `day` is the current day of `df` ;
- `df` is the given DataFrame ;
- `day_current_year` is the preceding day of the same year contained in `df_current_year`;
- `df_current_year` is the other given DataFrame, containing days of the same year.

The function returns `True` if and only if `current_year_day` is a day that has to be selected for *day*.

For a certain *day* of `df` it could happen that no preceding day of the same year is selected. This means that this *day* has a missing value for each of the *m* new columns (i.e. *m* missing values). In this case, if `replace_miss` is `True`, all the missing values are filled: the missing value for the new column *new_column* is filled computing the mean of all the preceding days

of *day* in the associated column in `df_current_year`. If there isn't any preceding day in `df_current_year`, that *day* is removed from the resulting DataFrame (the missing values can't be filled). (The removed days are surely the first days in `df`).
Otherwise, if `replace_miss` is `False`, the *m* missing values are simply kept as `Nan`. (No day has to be removed).

If `current_day` is `True`, each *day* of `df` is itself a potential same year day that can be selected. I.e. not only the preceding days are considered. (This is applied also in the selection of the days to be used to fill the missing values).

So, in the end, *m* new columns are created in the resulting DataFrame, from the selected *m* columns of `df_current_year`. From the selected column with name `"col"` of `df_current_year`, the corresponding column `"current_year_col"` is created in the resulting DataFrame.

In addition, the resulting DataFrame is automatically split into the X matrix and the y vector, which are respectively the matrix containing the explanatory features and the vector containing the response feature. (The response feature is the one which is the target of the prediction analysis tasks).

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- `df_current_year: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`). It should contain days of the same year with respect to the days in `df`.
- `days_to_select: int` or `str` or `callable`
  Indicates, for each day of `df`, which preceding days of the same year are selected in `df_current_year`. It must be either an integer or `"month"` or `"season"` or a predicate (i.e. a function that returns a boolean). The function signature must be
  ```
  (day: pd.TimeStamp, df: pd.DataFrame, day_current_year:
    pd.TimeStamp, df_current_year: pd.DataFrame): bool
  ```
- `current_day: bool`
  Indicates if each day of `df` can be potentially selected for itself as a day of the same year.
- `stat: str`
  Indicates the statistical aggregation to perform, for each day of `df`, on the selected same year days of `df_current_year`. It can either be `"mean"` or `"min"` or `"max"`.
- `columns_to_select: list`
  List of strings which indicates the columns of `df_current_year` that have to be taken into account. If it's `None`, all the columns of `df_current_year` are considered.
- `replace_miss: bool`
  Indicates whether to fill the missing values or keep them as `Nan`. (The missing

values are generated for each day of `df` for which no same year day in `df_current_year` is selected).

- `y_col: str`
  Indicates which is the column of the resulting DataFrame to be used as y column.
- `scale_y: bool`
  Indicates whether to scale or not the values of the response feature y.

Returns:

- `pd.DataFrame`
  The resulting DataFrame.
- `X: np.array`
  Two-dimensional numpy array which contains the explanatory features.
- `y: np.array`
  Mono-dimensional numpy array which contains the response feature.

Raises: `ValueError`

When `stat` is neither `"mean"` nor `"min"` nor `"max"`.

Warns: `UserWarning`

- When, for a day of `df`, no preceding day of the same year is selected from `df_current_year`.
- When, for a day of `df`, less preceding days of the same year are found compared to the ones expected. (This can happen only if `days_to_select` is either an integer or `"month"` or `"season"`).

-[Notes]-

If `add_current_year_statistics` is applied multiple times on the same `df` and `df_current_year`, columns with the same name are potentially created. For instance, if `add_current_year_statistics` is applied three times on the same DataFrames, from the `df_current_year` column `"col"` three different columns with the same name `"current_year_col"` are potentially created. To avoid that, `add_current_year_statistics` ensures that all the different columns with the same name are properly disambiguated, using progressive numbers. (E.g three different columns with the same name `"current_year_col"` became `"current_year_col"`, `"current_year_col.1"` and `"current_year_col.2"`).

```
def add_upTo_k_years_ago_statistics(df, df_upTo_k_years_ago, k=1,
current_year=True, days_to_select=11, current_day=False,
stat="mean", columns_to_select=None, replace_miss=False,
y_col=None, scale_y=True)
```

Add, to a time series DataFrame, statistics computed on the other given time series DataFrame, but with respect to the days of up to k years ago.

`df_upTo_k_years_ago` should contain days of up to k years ago with respect to the days of `df`.(Nevertheless, `df` can contain multiple years). Let *m* be the number of the selected columns of `df_upTo_k_years_ago` (by default all the columns, see the `columns_to_select` parameter). A new DataFrame is built, which is created from `df` adding *m* new columns. (The resulting DataFrame has the same index of `df`). These new *m* columns contain the values computed from the associated columns of `df_upTo_k_years_ago` considering the days of up to k years before the ones in `df`.

Let *day* be a row of `df`, and *new_column* be one of the *m* new columns created in the resulting DataFrame. The value put in that column for that day is computed from the associated column of `df_upTo_k_years_ago` considering the days of `df_upTo_k_years_ago` that are centered on `day` but up to k years ago.
Going into the details, for each integer *i* from 1 to k, the *i* years ago days centered on *day* and contained in `df_upTo_k_years_ago` are selected (see the `find_k_years_ago_days` function): from these selected *i* years ago days, an unique value is computed applying a certain statistical aggregation (specified by the input parameter `stat`) on the values of these selected days in the column of `df_upTo_k_years_ago` associated to *new_column*. In this way, for *day* and *new_column* k values are computed, for each integer *i* from 1 to k. In the end, an unique value for the new column *new_column* and for the day *day* is computed applying the same statistical aggregation (i.e. `stat`) on these k values. On the whole, an aggregation with 2 levels is computed on the days of up to k years ago with respect to *day*.

- An aggregation is computed on the selected days of *i* years ago, for *i* between 1 and k.
- An aggregation is computed on the k values computed for each of the previous years, up to k years ago.

Basically, this is implemented by applying k times the function `add_k_years_ago_statistics`: for each *i* between 1 and k, `add_k_years_ago_statistics` is applied with his input parameter *k* equal to *i*. `add_k_years_ago_statistics` is applied on all the previous years, up to k years ago. So, `add_upTo_k_years_ago_statistics` is nothing else than an extension of the `add_k_years_ago_statistics` function. (See `add_k_years_ago_statistics`). The meaning of the input parameters `days_to_select`, `stat`, `replace_miss`, ..., are the same of the ones seen in `add_k_years_ago_statistics`.

In particular, `days_to_select` specifies, for each *day* of `df`, which *i* years ago days in `df_upTo_k_years_ago` are selected, for *i* from 1 to k. It can either be an odd integer or `"month"` or `"season"` or a predicate (i.e. a function that returns a bool). The signature of the function must be:
`(day: pd.TimeStamp, df: pd.DataFrame, day_i_years_ago: pd.TimeStamp, df_upTo_k_years_ago : pd.DataFrame): bool.`

If `current_year` is `True`, also the current year is taken into account, and not only the preceding years up to k years ago. This means that, for each *day* of `df`, k+1 values are computed: from the current year; from the previous year; ...; from k years ago.
These k+1 values are aggregated in a single value. The value computed from the current year is calculated using the `add_current_year_statistics` function (see `add_current_year_statistics`). The meaning of the input parameters `days_to_select`, `current_day`, `stat`, `replace_miss`, ..., are the same as the ones seen in `add_current_year_statistics`.
(If `current_year` is `True`, `add_current_year_statistics` is applied one time and then `add_k_years_ago_statistics` is applied k times).

So, in the end, 'm' new columns are created in the resulting DataFrame, from the selected *m* columns of `df_upTo_k_years_ago`. From the selected column with name `"col"` of `df_upTo_k_years_ago`, the corresponding column `"upTo_k_years_ago_col"` is created in the resulting DataFrame.

In addition, the resulting DataFrame is automatically split into the X matrix and the y vector, which are respectively the matrix containing the explanatory features and the vector containing the response feature. (The response feature is the one which is the target of the prediction analysis tasks).

Parameters:

- `df: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`).
- `df_upTo_k_years_ago: pd.DataFrame`
  DataFrame indexed by days (i.e. the index is a `pd.DatetimeIndex`). It should contain up to k years ago days with respect to the days in `df`.
- `k: int`
  Indicates how many previous years have to be taken into account (i.e. all the previous years up to k years ago are taken into account). It must be a positive integer.
- `current_year: bool`
  Indicates whether to consider also the current year or not: in the former case are taken into account all the years from the current up to k years ago.
- `days_to_select: int` or `str` or `callable`
  Indicates, for each day of `df`, which days in `df_upTo_k_years_ago` have to be

selected, for each year from the previous up to k years ago.
It must either be an odd integer or "month" or "season" or a predicate (i.e. a function that returns a boolean).
If current_year is True, this selection is also applied on the days of the same year.

- current_day: bool
  Indicates if each day of df can be potentially selected for itself as a day of the same year. This parameter is considered only if current_year is True.
- stat: str
  Indicates the statistical aggregation to perform. It can either be "mean" or "min" or "max". This aggregation is applied in two levels: both for each previous year (up to k years ago) and for the aggregation of the k computed values (k+1 if current_year is True).
- columns_to_select: list
  List of strings which indicates the columns of df_upTo_k_years_ago that have to be taken into account. If it is None, all the columns of df_upTo_k_years_ago are considered.
- replace_miss: bool
  Indicates whether to fill the missing values.
- y_col: str
  Indicates which is the column of the resulting DataFrame to be used as y column.
- scale_y: bool
  Indicates whether to scale or not the values of the response feature y.

Returns:

- pd.DataFrame
  The resulting DataFrame.
- X: np.array
  Two-dimensional numpy array which contains the explanatory features.
- y: np.array
  Mono-dimensional numpy array which contains the response feature.

Raises: ValueError

- When k is not a positive integer.
- When stat is neither "mean" nor "min" nor "max".

-[Notes]-

- If add_upTo_k_years_ago_statistics is applied multiple times with the same k on the same df and df_upTo_k_years_ago, columns with the same name are potentially created. For instance, if add_upTo_k_years_ago_statistics is applied three times with the same k on the same DataFrames, from the df_upTo_k_years_ago column "col" three different columns with the same name "upTo_k_years_ago_col" are potentially created. To avoid that, add_upTo_k_years_ago_statistics ensures that all the different columns with the same name are properly disambiguated, using progressive numbers. (E.g three different columns with same name "upTo_k_years_ago_col" became "upTo_k_years_ago_col", "upTo_k_years_ago_col.1" and "upTo_k_years_ago_col.2").

- The meaning of `replace_miss` is the same seen in `add_k_years_ago_statistics`. For each previous year up to k years ago, if no selected day is found and `replace_miss` is `True`, the mean of the whole `df_upTo_k_years_ago` DataFrame is computed: this is the value calculated for that year (value that will be aggregated with the other `k values`). This same concept is valid also for the current year, if `current_year` is True.

EXAMPLES

```
>>> import timeSeries_processing as tsp
```

Prerequisites

In the examples shown in this document, the air pollution datasets of EEA will be used.

For this reason, the EEA-datasets-handler library will be utilized.
In particular, the PM10 mean concentrations in Italy are considered, with respect to 2020.

```
>>> import EEA_datasets_handler as eea

# Download the datasets
# IT'S NECESSARY ONLY IF THEY HAVEN'T BEEN DOWNLOADED YET
>>> dest_path = "C:\\Datasets"
>>> countries_cities_dict = {"IT": "all"}
>>> pollutants = ["PM10"]
>>> years = [2020]
>>> eea.download_datasets(dest_path, countries_cities_dict,
pollutants, years)

# Load the datasets
>>> source_path = "C:\\Datasets\\EEA"
>>> countries_cities_dict = {"IT":"all"}
>>> pollutants = ["PM10"]
>>> years = [2020]
>>> df = eea.load_datasets(source_path, countries_cities_dict,
pollutants, years)

# Process the datasets
>>> df_mean, _, _ = eea.preprocessing(df, fill=True,
fill_n_days=10 ,fill_aggr="mean")
UserWarning: Missing days: ['2020-01-30', '2020-01-31',
'2020-02-01', '2020-02-02', '2020-02-03', '2020-02-04',
'2020-02-05', '2020-02-06', '2020-02-07', '2020-02-08',
'2020-02-09', '2020-02-10', '2020-02-11']



>>> df_mean
                 mean
Datetime
2020-01-01  76.974569
2020-01-02  56.675791
2020-01-03  55.216906
2020-01-04  54.887035
2020-01-05  28.192059
...               ...
2020-12-27  14.997987
```

```
2020-12-28  16.317778
2020-12-29  23.536875
2020-12-30  22.759021
2020-12-31  22.005000

[366 rows x 1 columns]
```

Also other EEA datasets will be used later on in the following examples.
They will be loaded later, when required.

Finally, also the meteorological datasets of ILMETEO will be used.
In particular,the meteorological data of the whole Italy are considered, with respect to 2020.
These data are loaded from the local storage .
(See the appendix of the chapter about ILMETEO-datasets-handler).

```
>>> import pandas as pd
>>> df_meteo = pd.read_csv("meteorological_data_2020.csv",
index_col=0)
>>> df_meteo = df_meteo.set_index(pd.DatetimeIndex(df_meteo.index))

>>>  df_meteo
           TMEDIA °C   TMIN °C    TMAX °C  PUNTORUGIADA °C
2020-01-01   5.202196  0.173764  11.003131         2.534921
2020-01-02   4.774801  0.182640  10.325979         2.465371
2020-01-03   4.885063  0.531284   9.658314         3.231408
2020-01-04   6.162960  1.381113  11.354987         3.776887
2020-01-05   6.551925  1.686219  12.409042         3.172703
...               ...       ...        ...              ...
2020-12-27   3.366679  0.191966   6.335378         2.162273
2020-12-28   4.014919  1.075045   6.692360         3.125000
2020-12-29   4.921844  1.494424   8.174234         3.427775
2020-12-30   5.007944  2.084797   8.026350         3.539751
2020-12-31   3.614142 -0.334781   7.465027         2.430201

...
```

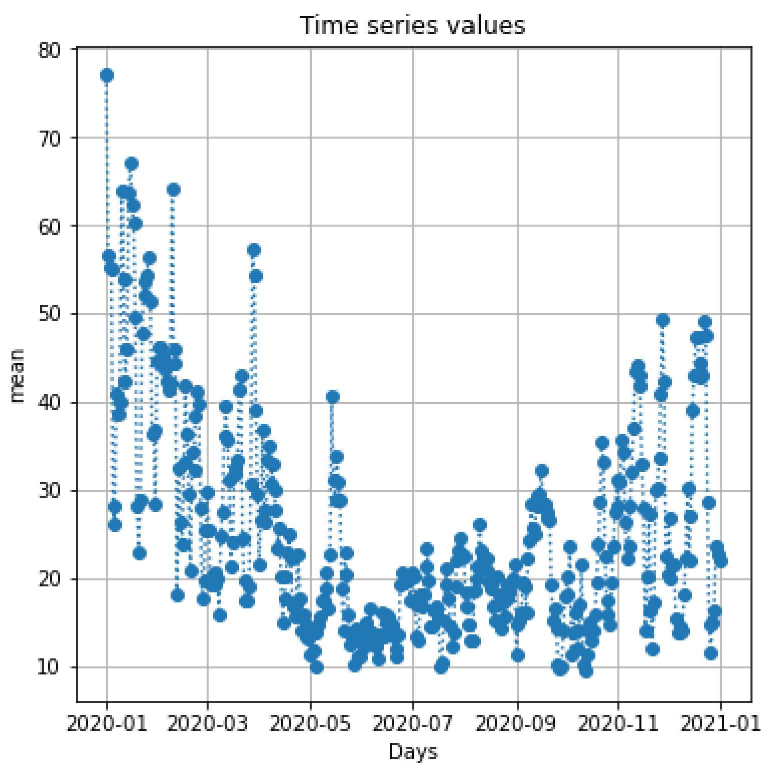
Functions to manipulate dates

Auxiliary functions which work with dates.

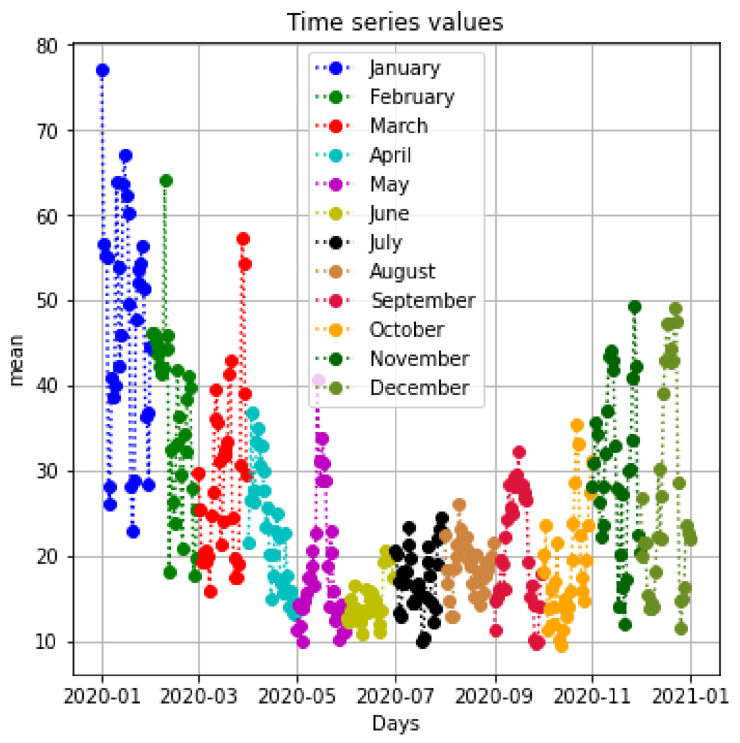
```
>>> import pandas as pd
>>> days = pd.date_range("2020-01-01",
"2020-01-04").append(pd.date_range("2020-01-11", "2020-03-14"))
>>> days
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03',
'2020-01-04', '2020-01-11', '2020-01-12', '2020-01-13',
'2020-01-14', '2020-01-15', '2020-01-16', '2020-01-17',
'2020-01-18', '2020-01-19', '2020-01-20', '2020-01-21',
'2020-01-22', '2020-01-23', '2020-01-24', '2020-01-25',
'2020-01-26', '2020-01-27', '2020-01-28', '2020-01-29',
'2020-01-30', '2020-01-31', '2020-02-01', '2020-02-02',
'2020-02-03', '2020-02-04', '2020-02-05', '2020-02-06',
'2020-02-07', '2020-02-08', '2020-02-09', '2020-02-10',
'2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14',
'2020-02-15', '2020-02-16', '2020-02-17', '2020-02-18',
'2020-02-19', '2020-02-20', '2020-02-21', '2020-02-22',
'2020-02-23', '2020-02-24', '2020-02-25', '2020-02-26',
'2020-02-27', '2020-02-28', '2020-02-29', '2020-03-01',
'2020-03-02', '2020-03-03', '2020-03-04', '2020-03-05',
'2020-03-06', '2020-03-07', '2020-03-08', '2020-03-09',
'2020-03-10', '2020-03-11', '2020-03-12', '2020-03-13',
'2020-03-14'],
dtype='datetime64[ns]', freq=None)

>>> tsp.find_missing_days(days)
DatetimeIndex(['2020-01-05', '2020-01-06', '2020-01-07',
'2020-01-08', '2020-01-09', '2020-01-10'],
dtype='datetime64[ns]', freq=None)

>>> tsp.group_days_by(days,criterion="year")
[('2020', DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03',
'2020-01-04', '2020-01-11', '2020-01-12', '2020-01-13',
'2020-01-14', '2020-01-15', '2020-01-16', '2020-01-17',
'2020-01-18', '2020-01-19', '2020-01-20', '2020-01-21',
'2020-01-22', '2020-01-23', '2020-01-24', '2020-01-25',
'2020-01-26', '2020-01-27', '2020-01-28', '2020-01-29',
'2020-01-30', '2020-01-31', '2020-02-01', '2020-02-02',
'2020-02-03', '2020-02-04', '2020-02-05', '2020-02-06',
'2020-02-07', '2020-02-08', '2020-02-09', '2020-02-10',
'2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14',
'2020-02-15', '2020-02-16', '2020-02-17', '2020-02-18',
'2020-02-19', '2020-02-20', '2020-02-21', '2020-02-22',
'2020-02-23', '2020-02-24', '2020-02-25', '2020-02-26',
'2020-02-27', '2020-02-28', '2020-02-29', '2020-03-01',
'2020-03-02', '2020-03-03', '2020-03-04', '2020-03-05',
'2020-03-06', '2020-03-07', '2020-03-08', '2020-03-09',
'2020-03-10', '2020-03-11', '2020-03-12', '2020-03-13',
'2020-03-14'],
dtype='datetime64[ns]', freq=None))]

>>> tsp.group_days_by(days,criterion="month")
[('January', DatetimeIndex(['2020-01-01', '2020-01-02',
'2020-01-03', '2020-01-04', '2020-01-11', '2020-01-12',
'2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16',
'2020-01-17', '2020-01-18', '2020-01-19', '2020-01-20',
'2020-01-21', '2020-01-22', '2020-01-23', '2020-01-24',
'2020-01-25', '2020-01-26', '2020-01-27', '2020-01-28',
'2020-01-29', '2020-01-30', '2020-01-31'],
 dtype='datetime64[ns]', freq=None)),
 ('February', DatetimeIndex(['2020-02-01', '2020-02-02',
'2020-02-03', '2020-02-04', '2020-02-05', '2020-02-06',
'2020-02-07', '2020-02-08', '2020-02-09', '2020-02-10',
'2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14',
'2020-02-15', '2020-02-16', '2020-02-17', '2020-02-18',
'2020-02-19', '2020-02-20', '2020-02-21', '2020-02-22',
'2020-02-23', '2020-02-24', '2020-02-25', '2020-02-26',
'2020-02-27', '2020-02-28', '2020-02-29'],
 dtype='datetime64[ns]', freq=None)),
('March', DatetimeIndex(['2020-03-01', '2020-03-02', '2020-03-03',
'2020-03-04', '2020-03-05', '2020-03-06', '2020-03-07',
'2020-03-08', '2020-03-09', '2020-03-10', '2020-03-11',
'2020-03-12', '2020-03-13', '2020-03-14'],
 dtype='datetime64[ns]', freq=None))]

>>> tsp.group_days_by(days,criterion="season")
[('Winter', DatetimeIndex(['2020-01-01', '2020-01-02',
'2020-01-03', '2020-01-04', '2020-01-11', '2020-01-12',
'2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16',
'2020-01-17', '2020-01-18', '2020-01-19', '2020-01-20',
'2020-01-21', '2020-01-22', '2020-01-23', '2020-01-24',
'2020-01-25', '2020-01-26', '2020-01-27', '2020-01-28',
'2020-01-29', '2020-01-30', '2020-01-31', '2020-02-01',
'2020-02-02', '2020-02-03', '2020-02-04', '2020-02-05',
'2020-02-06', '2020-02-07', '2020-02-08', '2020-02-09',
'2020-02-10', '2020-02-11', '2020-02-12', '2020-02-13',
'2020-02-14', '2020-02-15', '2020-02-16', '2020-02-17',
'2020-02-18', '2020-02-19', '2020-02-20', '2020-02-21',
'2020-02-22', '2020-02-23', '2020-02-24', '2020-02-25',
'2020-02-26', '2020-02-27', '2020-02-28', '2020-02-29'],
dtype='datetime64[ns]', freq=None)),
('Spring', DatetimeIndex(['2020-03-01', '2020-03-02',
'2020-03-03', '2020-03-04', '2020-03-05', '2020-03-06',
'2020-03-07', '2020-03-08', '2020-03-09', '2020-03-10',
'2020-03-11', '2020-03-12', '2020-03-13', '2020-03-14'],
 dtype='datetime64[ns]', freq=None))]
```

Function to plot a time series

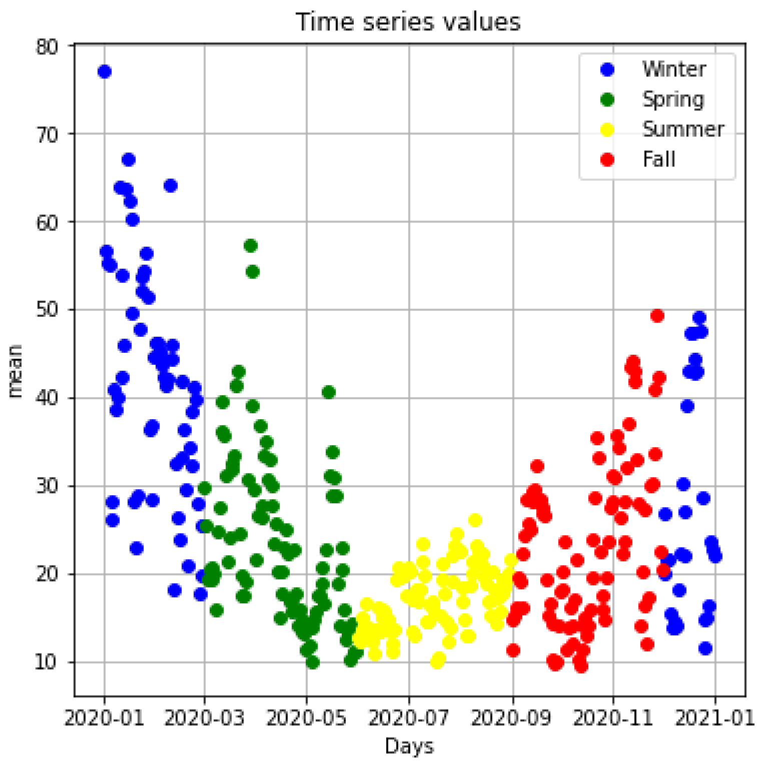
```
>>> tsp.plot_timeSeries(df_mean, col_name="mean", figsize=(6,6))
<AxesSubplot:title={'center':'Time series values'}, xlabel='Days',
ylabel='mean'>
```



```
>>> tsp.plot_timeSeries(df_mean, col_name="mean", divide="month",
figsize=(6,6))
<AxesSubplot:title={'center':'Time series values'}, xlabel='Days',
ylabel='mean'>
```



```
>>> tsp.plot_timeSeries(df_mean, col_name="mean", divide="season",
line=False, figsize=(6,6))
<AxesSubplot:title={'center':'Time series values'}, xlabel='Days',
ylabel='mean'>
```



add_timeSeries_dataframe

```
>>> df_meteo = df_meteo[["TMEDIA °C","TMIN °C"]] # Focus on two
meteorological features
>>> df_mean_met, X, y = tsp.add_timeSeries_dataframe(df=df_mean,
df_other=df_meteo, y_col="mean")
>>> df_mean_met
                mean  TMEDIA °C   TMIN °C
Datetime
2020-01-01  76.974569   5.202196  0.173764
2020-01-02  56.675791   4.774801  0.182640
2020-01-03  55.216906   4.885063  0.531284
2020-01-04  54.887035   6.162960  1.381113
2020-01-05  28.192059   6.551925  1.686219
...               ...        ...       ...
```

```
2020-12-27  14.997987   3.366679  0.191966
2020-12-28  16.317778   4.014919  1.075045
2020-12-29  23.536875   4.921844  1.494424
2020-12-30  22.759021   5.007944  2.084797
2020-12-31  22.005000   3.614142 -0.334781

[366 rows x 3 columns]
```

The `add_timeSeries_dataframe` function, like all the other following processing functions, also returns the X and y numpy arrays.

```
>>> X # Contains the explanatory features
array([[ 5.20219576,  0.17376411],
       [ 4.77480103,  0.18264044],
       [ 4.88506323,  0.5312842 ],
       [ 6.16295977,  1.38111267],
       [ 6.55192451,  1.68621891],
       [ 4.55391366,  0.61115109],
       [ 2.62864489, -0.99138223],
       [ 3.7207561 , -0.42611325],
       [ 4.26103639, -0.71598666],
       [ 5.5886528 ,  0.60886076],
       [ 5.76592089,  0.47124603],
       [ 5.47536483,  1.03039838],
       [ 4.2906024 , -0.24437726],
       [ 4.30190439, -0.04794869],
       [ 5.44510059,  1.82439534],
…])

>>> y # Contains the response feature. It has been scaled
array([1.        , 0.69936896, 0.67776243, 0.67287694, 0.27751628,
       0.24652335, 0.46364727, 0.4323436 , 0.450411  , 0.80705539,
       0.6582936 , 0.48497419, 0.53898003, 0.80161642, 0.85440625,
       0.78269998, 0.7518062 , 0.59348505, 0.27736707, 0.20026144,
       0.28810984, 0.56534763, 0.65398233, 0.63206399, 0.6642047 ,
       0.69506819, 0.62197136, 0.39715901, 0.40522439, 0.28102485,
       0.52041563, 0.54364621, 0.54147607, 0.53022544, 0.52004159,
       0.50562527, 0.48668098, 0.47315194, 0.48075124, 0.80784271,
       0.54098571, 0.51430001, 0.12899232, 0.34166921, 0.25098551,
       0.2118916 , 0.35118668, 0.47911109, 0.39750183, 0.2959135 ,
       0.16996786, 0.36619306, 0.3363639 , 0.42957547, 0.46857668,
       0.44812558, 0.27374897, 0.12216556, 0.15038488, 0.23601898,
…])
```

With the input parameter `y_scale`*, the user can decide whether to scale or not the response feature.*

add_k_previous_days

```
>>> df_mean_temp, X, y = tsp.add_k_previous_days(df=df_mean,
col_name="mean", k=5, y_col="mean")
>>> df_mean_temp.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 361 entries, 2020-01-06 to 2020-12-31
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   mean    361 non-null    float64
 1   mean_1  361 non-null    float64
 2   mean_2  361 non-null    float64
 3   mean_3  361 non-null    float64
 4   mean_4  361 non-null    float64
 5   mean_5  361 non-null    float64
dtypes: float64(6)
memory usage: 19.7 KB

>>> df_mean_temp
                 mean     mean_1     mean_2     mean_3     mean_4
Datetime
2020-01-06  26.099399  28.192059  54.887035  55.216906  56.675791
2020-01-07  40.759729  26.099399  28.192059  54.887035  55.216906
2020-01-08  38.646087  40.759729  26.099399  28.192059  54.887035
2020-01-09  39.866008  38.646087  40.759729  26.099399  28.192059
2020-01-10  63.946840  39.866008  38.646087  40.759729  26.099399
...               ...        ...        ...        ...        ...
2020-12-27  14.997987  11.480358  14.695619  28.473515  47.568209
2020-12-28  16.317778  14.997987  11.480358  14.695619  28.473515
2020-12-29  23.536875  16.317778  14.997987  11.480358  14.695619
2020-12-30  22.759021  23.536875  16.317778  14.997987  11.480358
2020-12-31  22.005000  22.759021  23.536875  16.317778  14.997987
```

```
                mean_5
Datetime
2020-01-06  76.974569
2020-01-07  56.675791
2020-01-08  55.216906
2020-01-09  54.887035
2020-01-10  28.192059
...               ...
2020-12-27  49.143828
2020-12-28  47.568209
2020-12-29  28.473515
2020-12-30  14.695619
2020-12-31  11.480358

[361 rows x 6 columns]
```

The names of the added columns have the structure "`ColumnName_i`", with i from 1 to k.

add_k_years_ago_statistics

Get the EEA datasets about the PM10 mean concentrations in Italy, with respect to 2019 (i.e. one year before 2020).

```
# Download the datasets
# IT'S NECESSARY ONLY IF THEY HAVEN'T BEEN DOWNLOADED YET
>>> dest_path = "C:\\Datasets"
>>> countries_cities_dict = {"IT": "all"}
>>> pollutants = ["PM10"]
>>> years = [2019]
>>> eea.download_datasets(dest_path, countries_cities_dict,
pollutants, years)

# Load the datasets
>>> source_path = "C:\\Datasets\\EEA"
>>> countries_cities_dict = {"IT": "all"}
>>> pollutants = ["PM10"]
>>> years = [2019]
>>> df_prev_year = eea.load_datasets(source_path,
countries_cities_dict, pollutants, years)

# Process the datasets
>>> df_prev_year_mean, df_prev_year_min, df_prev_year_max =
eea.preprocessing(df_prev_year)
```

First example: `days_to_select` is an odd integer

```
>>>df_mean_temp, X, y = tsp.add_k_years_ago_statistics(df=df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=11,
y_col="mean")
UserWarning: For the day 2020-01-01 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06']
UserWarning: For the day 2020-01-02 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07']
...
UserWarning: For the day 2020-12-27 only these 1 years ago days
have been found: ['2019-12-22', '2019-12-23', '2019-12-24',
'2019-12-25', '2019-12-26', '2019-12-27', '2019-12-28',
'2019-12-29', '2019-12-30', '2019-12-31']
...
```

A warning is given:
- for each day for which less days than the ones expected are selected;
- for each day for which no day has been selected.

```
>>> df_mean_temp
                mean  1_years_ago_mean
Datetime
2020-01-01  76.974569         26.371291
2020-01-02  56.675791         26.875353
2020-01-03  55.216906         28.239070
2020-01-04  54.887035         28.412754
2020-01-05  28.192059         27.278523
...               ...               ...
```

```
2020-12-27  14.997987         24.077666
2020-12-28  16.317778         25.042372
2020-12-29  23.536875         25.884439
2020-12-30  22.759021         25.940512
2020-12-31  22.005000         25.486811

[366 rows x 2 columns]
```

The names of the added columns have the structure `"k_years_ago_ColumnName"`.

In order to compute the value for a 2020 day, the 11 days of 2019 that are centered on that day are selected.

```
# 2020-04-15
>>> df_mean_temp["1_years_ago_mean"].loc[pd.Timestamp("2020-04-15")]
18.504581483793103
>>> df_prev_year_mean["mean"].loc[pd.date_range('2019-04-10',
'2019-04-20')].mean()
18.504581483793103

# 2020-10-22
>>> df_mean_temp["1_years_ago_mean"].loc[pd.Timestamp("2020-10-22")]
26.68901718805421
>>> df_prev_year_mean["mean"].loc[pd.date_range('2019-10-17',
'2019-10-27')].mean()
26.68901718805421

# 2020-01-01: example with less than 11 days
>>> df_mean_temp["1_years_ago_mean"].loc[pd.Timestamp("2020-01-01")]
26.37129108442753
>>> df_prev_year_mean["mean"].loc[pd.date_range('2019-01-01',
'2019-01-06')].mean()
26.37129108442753

# 2020-12-27: example with less than 11 days
>>> df_mean_temp["1_years_ago_mean"].loc[pd.Timestamp("2020-12-27")]
24.0776657907545
>>> df_prev_year_mean["mean"].loc[pd.date_range('2019-12-22',
'2019-12-31')].mean()
24.0776657907545
```

With the input parameter `scale`*, the user can decide which is the statistical aggregation to be used.*

`days_to_select` is `"month"` (/`"season"`)

```
>>>df_mean_temp, _, _ = tsp.add_k_years_ago_statistics(df=df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select="month",
y_col="mean")
>>> df_mean_temp
                 mean  1_years_ago_mean
Datetime
2020-01-01  76.974569         25.813125
2020-01-02  56.675791         25.813125
2020-01-03  55.216906         25.813125
2020-01-04  54.887035         25.813125
2020-01-05  28.192059         25.813125
...               ...               ...
2020-12-27  14.997987         26.776342
2020-12-28  16.317778         26.776342
2020-12-29  23.536875         26.776342
2020-12-30  22.759021         26.776342
2020-12-31  22.005000         26.776342

[366 rows x 2 columns]
```

In order to compute the value for a 2020 day, the days of the same month in 2019 are selected.

```
# January
>>> df_prev_year_mean["mean"].loc[pd.date_range('2019-01-01',
'2019-01-31')].mean()
25.813124974351396
```

More columns

The same logic is applied individually on each column of `df_prev_year_mean`.

```
# Put a second column
>>> df_prev_year_mean["COLUMN"] = df_prev_year_mean["mean"]*2
>>> df_mean_temp, _ ,_ = tsp.add_k_years_ago_statistics(df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=11,
y_col="mean")
UserWarning: For the day 2020-01-01 only these 1 years ago days
```

```
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06']
UserWarning: For the day 2020-01-02 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07']
...
>>> df_mean_temp
                mean  1_years_ago_mean  1_years_ago_COLUMN
Datetime
2020-01-01  76.974569         26.371291           52.742582
2020-01-02  56.675791         26.875353           53.750707
2020-01-03  55.216906         28.239070           56.478140
2020-01-04  54.887035         28.412754           56.825508
2020-01-05  28.192059         27.278523           54.557047
...               ...               ...                 ...
2020-12-27  14.997987         24.077666           48.155332
2020-12-28  16.317778         25.042372           50.084744
2020-12-29  23.536875         25.884439           51.768877
2020-12-30  22.759021         25.940512           51.881024
2020-12-31  22.005000         25.486811           50.973623

[366 rows x 3 columns]
```

With the input parameter `columns_to_select`, the user can specify which are the columns to be taken into account.

```
>>> df_mean_temp, _, _ = tsp.add_k_years_ago_statistics(df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=11,
columns_to_select=["mean"], y_col="mean")
UserWarning: For the day 2020-01-01 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06']
UserWarning: For the day 2020-01-02 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07']
...
>>> df_mean_temp
                mean  1_years_ago_mean
Datetime
2020-01-01  76.974569         26.371291
```

```
2020-01-02  56.675791          26.875353
2020-01-03  55.216906          28.239070
2020-01-04  54.887035          28.412754
2020-01-05  28.192059          27.278523
...               ...                ...
2020-12-27  14.997987          24.077666
2020-12-28  16.317778          25.042372
2020-12-29  23.536875          25.884439
2020-12-30  22.759021          25.940512
2020-12-31  22.005000          25.486811

[366 rows x 2 columns]
```

`days_to_select` is a function

It's a predicate that decides which days have to be selected

```
# Predicate of selection
>>> f = lambda day, df, prev_day, prev_df :
abs(df["mean"][day]-prev_df["mean"][prev_day])<3

>>>df_mean_temp, _, _ = tsp.add_k_years_ago_statistics(df=df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=f,
y_col="mean")
UserWarning: No 1 years ago days have been found for the day
2020-01-01
UserWarning: No 1 years ago days have been found for the day
2020-01-16
>>> df_mean_temp
                 mean  1_years_ago_mean  1_years_ago_COLUMN
Datetime
2020-01-01  76.974569          23.034827           46.069654
2020-01-02  56.675791          56.540150          113.080300
2020-01-03  55.216906          55.693172          111.386345
2020-01-04  54.887035          55.693172          111.386345
2020-01-05  28.192059          27.994942           55.989884
...               ...                ...                 ...
2020-12-27  14.997987          15.388949           30.777898
2020-12-28  16.317778          16.423637           32.847274
2020-12-29  23.536875          23.249990           46.499979
```

```
2020-12-30  22.759021         22.684613           45.369227
2020-12-31  22.005000         21.775507           43.551014

[366 rows x 3 columns]
```

For each 2020 day, the 2019 days with similar PM10 concentration are selected (i.e. the 2019 days whose PM10 concentration differ from the PM10 concentration of the 2020 day less than 3).

replace_miss

By default, the parameter `replace_miss` is `True`. I.e. the missing days are filled.

```
# replace_miss True
>>> f = lambda day,df,prev_day,prev_df :
abs(df["mean"][day]-prev_df["mean"][prev_day])<3

>>>df_mean_temp, _, _ = tsp.add_k_years_ago_statistics(df=df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=f,
replace_miss=True, y_col="mean")
UserWarning: No 1 years ago days have been found for the day
2020-01-01
UserWarning: No 1 years ago days have been found for the day
2020-01-16

>>> df_mean_temp
                mean  1_years_ago_mean  1_years_ago_COLUMN
Datetime
2020-01-01  76.974569         23.034827           46.069654
2020-01-02  56.675791         56.540150          113.080300
2020-01-03  55.216906         55.693172          111.386345
2020-01-04  54.887035         55.693172          111.386345
2020-01-05  28.192059         27.994942           55.989884
...               ...               ...                 ...
2020-12-27  14.997987         15.388949           30.777898
2020-12-28  16.317778         16.423637           32.847274
2020-12-29  23.536875         23.249990           46.499979
2020-12-30  22.759021         22.684613           45.369227
2020-12-31  22.005000         21.775507           43.551014

[366 rows x 3 columns]
```

The 2020-01-01 is a missing day, and it has been filled.

```
# replace_miss False
>>> f = lambda day,df,prev_day,prev_df :
abs(df["mean"][day]-prev_df["mean"][prev_day])<3

>>>df_mean_temp, _, _ = tsp.add_k_years_ago_statistics(df=df_mean,
df_k_years_ago=df_prev_year_mean, k=1, days_to_select=f,
replace_miss=False, y_col="mean")
UserWarning: No 1 years ago days have been found for the day
2020-01-01
UserWarning: No 1 years ago days have been found for the day
2020-01-16

>>> df_mean_temp
                mean  1_years_ago_mean  1_years_ago_COLUMN
Datetime
2020-01-01  76.974569               NaN                 NaN
2020-01-02  56.675791         56.540150          113.080300
2020-01-03  55.216906         55.693172          111.386345
2020-01-04  54.887035         55.693172          111.386345
2020-01-05  28.192059         27.994942           55.989884
...               ...               ...                 ...
2020-12-27  14.997987         15.388949           30.777898
2020-12-28  16.317778         16.423637           32.847274
2020-12-29  23.536875         23.249990           46.499979
2020-12-30  22.759021         22.684613           45.369227
2020-12-31  22.005000         21.775507           43.551014

[366 rows x 3 columns]
```

add_current_year_statistics

First example: `days_to_select` is an integer.

```
>>> df_mean_temp, _, _ = tsp.add_current_year_statistics(df_mean,
df_current_year=df_mean, days_to_select=11, y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: For the day 2020-01-02 only these current year days
have been found: ['2020-01-01']
UserWarning: For the day 2020-01-03 only these current year days
have been found: ['2020-01-01', '2020-01-02']
For the day 2020-01-04 only these current year days have been
found: ['2020-01-01', '2020-01-02', '2020-01-03']
...
```

The same warnings of the previous function are given.

```
>>> df_mean_temp
                mean  current_year_mean
Datetime
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          66.825180
2020-01-04  54.887035          62.955755
2020-01-05  28.192059          60.938575
2020-01-06  26.099399          54.389272
...               ...                ...
2020-12-27  14.997987          38.072739
2020-12-28  16.317778          35.527418
2020-12-29  23.536875          32.718678
2020-12-30  22.759021          30.565353
2020-12-31  22.005000          28.598067

[365 rows x 2 columns]
```

The names of the added columns have the structure `"current_year_ColumnName"`.

In order to compute the value for a day, the 11 days preceding that day are selected.

```
# 2020-04-15
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-04-15")]
28.44221962405437
>>> df_mean["mean"].loc[pd.date_range('2020-04-04',
'2020-04-14')].mean()
28.44221962405437

# 2020-10-22
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-10-22")]
15.826564427994192
```

```
>>> df_mean["mean"].loc[pd.date_range('2020-10-11',
'2020-10-21')].mean()
15.826564427994192

# 2020-01-03: example with less than 11 days
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-03")]
66.82518027076988
>>> df_mean["mean"].loc[pd.date_range('2020-01-01',
'2020-01-02')].mean()
66.82518027076988

# 2020-01-05: example with less than 11 days
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-05")]
60.938575393321
>>> df_mean["mean"].loc[pd.date_range('2020-01-01',
'2020-01-04')].mean()
60.938575393321
```

With the input parameter `scale`, the user can decide which is the statistical aggregation to be used.

replace_miss

By default, the parameter `replace_miss` is `True`. I.e. the missing days are filled.
The days for which no preceding day is found are deleted from the dataset.

```
>>> df_mean_temp, _, _ = tsp.add_current_year_statistics(df_mean,
df_current_year=df_mean, days_to_select=11, y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: For the day 2020-01-02 only these current year days
have been found: ['2020-01-01']
UserWarning: For the day 2020-01-03 only these current year days
have been found: ['2020-01-01', '2020-01-02']
For the day 2020-01-04 only these current year days have been
found: ['2020-01-01', '2020-01-02', '2020-01-03']
...
>>> df_mean_temp
                mean  current_year_mean
Datetime
2020-01-02  56.675791          76.974569
```

```
2020-01-03  55.216906          66.825180
2020-01-04  54.887035          62.955755
2020-01-05  28.192059          60.938575
2020-01-06  26.099399          54.389272
...               ...                ...
2020-12-27  14.997987          38.072739
2020-12-28  16.317778          35.527418
2020-12-29  23.536875          32.718678
2020-12-30  22.759021          30.565353
2020-12-31  22.005000          28.598067

[365 rows x 2 columns]
```
No preceding day is present for the day 2020-01-01: this day has been removed.

If `replace_miss` is `False`, the missing values are kept as Nan.

```
>>> df_mean_temp, _, _ = tsp.add_current_year_statistics(df_mean,
df_current_year=df_mean, days_to_select=11, replace_miss=False,
y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: For the day 2020-01-02 only these current year days
have been found: ['2020-01-01']
UserWarning: For the day 2020-01-03 only these current year days
have been found: ['2020-01-01', '2020-01-02']
For the day 2020-01-04 only these current year days have been
found: ['2020-01-01', '2020-01-02', '2020-01-03']
...
>>> df_mean_temp
                 mean  current_year_mean
Datetime
2020-01-01  76.974569                NaN
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          66.825180
2020-01-04  54.887035          62.955755
2020-01-05  28.192059          60.938575
...               ...                ...
2020-12-27  14.997987          38.072739
2020-12-28  16.317778          35.527418
2020-12-29  23.536875          32.718678
2020-12-30  22.759021          30.565353
```

```
2020-12-31   22.005000          28.598067

[366 rows x 2 columns]
```

`current_day`

By default the parameter `current_day` is `False`. I.e. each day is not itself selected, but only the preceding days are selected.
Otherwise, if it's `True`, each day is itself selected.

```
>>> df_mean_temp,_,_ = tsp.add_current_year_statistics(df_mean,
df_current_year=df_mean, days_to_select=11, current_day=True,
y_col="mean")
UserWarning: For the day 2020-01-01 only these current year days
have been found: ['2020-01-01']
UserWarning: For the day 2020-01-02 only these current year days
have been found: ['2020-01-01', '2020-01-02']
UserWarning: For the day 2020-01-03 only these current year days
have been found: ['2020-01-01', '2020-01-02', '2020-01-03']
UserWarning: For the day 2020-01-04 only these current year days
have been found: ['2020-01-01', '2020-01-02', '2020-01-03',
'2020-01-04']
...
>>> df_mean_temp
                mean   current_year_mean
Datetime
2020-01-01   76.974569          76.974569
2020-01-02   56.675791          66.825180
2020-01-03   55.216906          62.955755
2020-01-04   54.887035          60.938575
2020-01-05   28.192059          54.389272
...                ...                ...
2020-12-27   14.997987          35.527418
2020-12-28   16.317778          32.718678
2020-12-29   23.536875          30.565353
2020-12-30   22.759021          28.598067
2020-12-31   22.005000          26.725684

[366 rows x 2 columns]
```

In order to compute the value for a day, the day itself and the 10 days preceding that day are selected.

```
# 2020-04-15
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-04-15")]
27.28543001471924
>>> df_mean["mean"].loc[pd.date_range('2020-04-05',
'2020-04-15')].mean()
27.28543001471924

# 2020-10-22
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-10-22")]
18.122953462767608
>>> df_mean["mean"].loc[pd.date_range('2020-10-12',
'2020-10-22')].mean()
18.122953462767608

# 2020-01-03: example with less than 11 days
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-03")]
62.95575545599534
>>> df_mean["mean"].loc[pd.date_range('2020-01-01',
'2020-01-03')].mean()
62.95575545599534

# 2020-01-05: example with less than 11 days
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-05")]
54.38927215262471
>>> df_mean["mean"].loc[pd.date_range('2020-01-01',
'2020-01-05')].mean()
54.38927215262471
```

It's important to notice that, despite the fact that `replcae_miss` is `True`, the day 2020-01-01 hasn't been removed: this is because it isn't a missing day anymore (it has been selected at least one day, which is the day itself).

`days_to_select` is `"month"` (/`"season"`)

```
>>> df_mean_temp, _, _ = tsp.add_current_year_statistics(df_mean,
df_current_year=df_mean, days_to_select="month", y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: No current year days have been found for the day
```

```
2020-02-01
UserWarning: No current year days have been found for the day
2020-03-01
UserWarning: No current year days have been found for the day
2020-04-01
...
>>> df_mean_temp
                 mean  current_year_mean
Datetime
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          66.825180
2020-01-04  54.887035          62.955755
2020-01-05  28.192059          60.938575
2020-01-06  26.099399          54.389272
...               ...                ...
2020-12-27  14.997987          28.392754
2020-12-28  16.317778          27.896652
2020-12-29  23.536875          27.483120
2020-12-30  22.759021          27.347043
2020-12-31  22.005000          27.194109

[365 rows x 2 columns]
```

In order to compute the value for a day, the preceding days that are in the same month are selected.

```
# 2020-04-15
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-04-15")]
28.405827728851147
>>> df_mean["mean"].loc[pd.date_range('2020-04-01',
'2020-04-14')].mean()
28.405827728851147

# 2020-10-22
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-10-22")]
16.255132480170204
>>> df_mean["mean"].loc[pd.date_range('2020-10-01',
'2020-10-21')].mean()
16.255132480170204
```

If `curret_day` is True, the day itself is also taken.

`days_to_select` is a function

It's a predicate that decides which preceding days have to be selected. (*If `curret_day` is True, the day itself can also be taken*).

```
# Predicate of selection
>>> f = lambda day, df, current_day ,current_df:
abs(df["mean"].loc[day]-current_df["mean"].loc[current_day])<3

>>> df_mean_temp,_,_ = tsp.add_current_year_statistics(df=df_mean,
df_current_year=df_mean, days_to_select=f, y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: No current year days have been found for the day
2020-01-02
UserWarning: No current year days have been found for the day
2020-01-05
UserWarning: No current year days have been found for the day
2020-01-07
UserWarning: No current year days have been found for the day
2020-01-10
...

>>> df_mean_temp
                mean  current_year_mean
Datetime
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          56.675791
2020-01-04  54.887035          55.946348
2020-01-05  28.192059          60.938575
2020-01-06  26.099399          28.192059
...               ...                ...
2020-12-27  14.997987          15.034873
2020-12-28  16.317778          16.110946
2020-12-29  23.536875          23.132278
2020-12-30  22.759021          22.102458
2020-12-31  22.005000          21.409305

[365 rows x 2 columns]
```

For each day, the preceding days with similar PM10 concentration are selected (i.e. the preceding days whose PM10 concentration differ from the PM10 concentration of the current day less than 3).

```
>>> df_mean[:10]
                 mean
Datetime
2020-01-01  76.974569
2020-01-02  56.675791
2020-01-03  55.216906
2020-01-04  54.887035
2020-01-05  28.192059
2020-01-06  26.099399
2020-01-07  40.759729
2020-01-08  38.646087
2020-01-09  39.866008
2020-01-10  63.946840

# 2020-01-04: 2020-01-03 and 2020-01-02 are selected
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-04")]
55.94634849921039
>>> (56.675791+55.216906)/2
55.94634849921039

# 2020-01-09: 2020-01-08 and 2020-01-07 are selected
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-09")]
39.70290812026687
>>> (40.759729+38.646087)/2
39.70290812026687

# 2020-01-05: is a missing day. The miss value is filled taking into
account all the previous days.
>>>df_mean_temp["current_year_mean"].loc[pd.Timestamp("2020-01-05")]
60.938575393321
>>> (76.974569+56.675791+55.216906+54.887035)/4
60.938575393321
```

More columns

The same logic is applied individually for each column of `df_current_year`. With the parameter `columns_to_select` the user can specify which are the columns that have to be used.
This is shown in the examples for the previous function : `add_k_years_ago_statistics`.

Example with meteorological data

Dataset with both the PM10 mean concentrations and the meteorological features

```
>>> df_mean_met,_,_ = tsp.add_timeSeries_dataframe(df_mean,
df_meteo, y_col="mean")
>>> df_mean_met
                mean  TMEDIA °C   TMIN °C    TMAX °C
Datetime
2020-01-01  76.974569   5.202196  0.173764  11.003131
2020-01-02  56.675791   4.774801  0.182640  10.325979
2020-01-03  55.216906   4.885063  0.531284   9.658314
2020-01-04  54.887035   6.162960  1.381113  11.354987
2020-01-05  28.192059   6.551925  1.686219  12.409042
...               ...        ...       ...        ...
2020-12-27  14.997987   3.366679  0.191966   6.335378
2020-12-28  16.317778   4.014919  1.075045   6.692360
2020-12-29  23.536875   4.921844  1.494424   8.174234
2020-12-30  22.759021   5.007944  2.084797   8.026350
2020-12-31  22.005000   3.614142 -0.334781   7.465027


...

[366 rows x 13 columns]
```

```
>>> f = lambda day,df,d,df_current_day: abs(df_current_day["TMEDIA
°C"].loc[day]-df_current_day["TMEDIA °C"].loc[d])<0.5

>>> df_mean_temp,_,_ = tsp.add_current_year_statistics(df=df_mean,
df_current_year=df_mean_met, days_to_select=f,
replace_miss=False, # Miss values are not replaced
columns_to_select=["mean"], # Only the column "mean" is taken into
account
y_col="mean")
UserWarning: No current year days have been found for the day
2020-01-01
UserWarning: No current year days have been found for the day
2020-01-04
UserWarning: No current year days have been found for the day
```

```
2020-01-07
UserWarning: No current year days have been found for the day
2020-01-08
UserWarning: No current year days have been found for the day
2020-01-28
UserWarning: No current year days have been found for the day
2020-02-03
...

>>> df_mean_temp
                 mean   current_year_mean
Datetime
2020-01-01  76.974569                NaN
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          66.825180
2020-01-04  54.887035                NaN
2020-01-05  28.192059          54.887035
...               ...                ...
2020-12-27  14.997987          38.646087
2020-12-28  16.317778          46.984483
2020-12-29  23.536875          39.790746
2020-12-30  22.759021          39.844928
2020-12-31  22.005000          23.320618

[366 rows x 2 columns]
```

For each day, the preceding days which have similar mean temperatures are selected (i.e. the difference between the temperatures is less than 0.5).

```
>>> df_mean_met[["TMEDIA °C"]][:10]
           TMEDIA °C
Datetime
2020-01-01   5.202196
2020-01-02   4.774801
2020-01-03   4.885063
2020-01-04   6.162960
2020-01-05   6.551925
2020-01-06   4.553914
2020-01-07   2.628645
2020-01-08   3.720756
2020-01-09   4.261036
```

```
2020-01-10   5.588653
>>> df_mean_temp[:10]
                mean  current_year_mean
Datetime
2020-01-01  76.974569                NaN
2020-01-02  56.675791          76.974569
2020-01-03  55.216906          66.825180
2020-01-04  54.887035                NaN
2020-01-05  28.192059          54.887035
2020-01-06  26.099399          55.946348
2020-01-07  40.759729                NaN
2020-01-08  38.646087                NaN
2020-01-09  39.866008          26.099399
2020-01-10  63.946840          76.974569

# 2020-01-03: 2020-01-02 and 2020-01-01 are selected
>>> (56.675791+76.974569)/2

# 2020-01-05: only 2020-01-04 is selected
>>> 54.887035

# 2020-01-09: only 2020-01-06 is selected
>>> 26.099399
```

add_upTo_k_years_ago_statistics

Get the EEA datasets about the PM10 mean concentrations in Italy, with respect to 2018-2019-2020 (i.e. up to two years before 2020).

```
# Download the datasets
# IT'S NECESSARY ONLY IF THEY HAVEN'T BEEN DOWNLOADED YET
>>> dest_path = "C:\\Datasets"
>>> countries_cities_dict = {"IT": "all"}
>>> pollutants = ["PM10"]
>>> years = [2018, 2019, 2020]
>>> eea.download_datasets(dest_path, countries_cities_dict,
pollutants, years)

# Load the datasets
>>> source_path = "C:\\Datasets\\EEA"
>>> countries_cities_dict = {"IT":"all"}
>>> pollutants = ["PM10"]
>>> years = [2018, 2019, 2020]
>>> df_full = eea.load_datasets(source_path,
countries_cities_dict, pollutants, years)

# Processing
>>> df_full_mean, df_full_min, df_full_max =
eea.preprocessing(df_full)
UserWarning: Missing days: ['2020-01-31', '2020-02-01',
'2020-02-02', '2020-02-03', '2020-02-04', '2020-02-05',
'2020-02-06', '2020-02-07', '2020-02-08', '2020-02-10',
'2020-02-11']
```

```
>>>df_mean_temp,_,_ = tsp.add_upTo_k_years_ago_statistics(df_mean,
df_upTo_k_years_ago=df_full_mean, k=2, days_to_select=11,
y_col="mean")
UserWarning: For the day 2020-01-01 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06']
UserWarning: For the day 2020-01-02 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07']
UserWarning: For the day 2020-01-03 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07',
'2019-01-08']
UserWarning: For the day 2020-01-04 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07',
'2019-01-08', '2019-01-09']
…

>>> df_mean_temp
               mean   upTo_2_years_ago_mean
Datetime
```

```
2020-01-01  76.974569                24.944020
2020-01-02  56.675791                27.880914
2020-01-03  55.216906                30.439158
2020-01-04  54.887035                32.203776
2020-01-05  28.192059                32.972802
...               ...                      ...
2020-12-27  14.997987                33.063248
2020-12-28  16.317778                31.808265
2020-12-29  23.536875                30.577231
2020-12-30  22.759021                30.369128
2020-12-31  22.005000                30.312998

[366 rows x 2 columns]
```

The names of the added columns have the structure "`upTo_k_years_ago_ColumnName`".

Basically, the function `add_k_years_ago_statistics` is applied for each of the specified previous years (i.e. for 2019 and for 2018) and then the mean of the computed values is calculated: in this way, for each day of 2020 an unique value is obtained, which sums up the two previous years.

The semantics of the parameters `days_to_select`, `stat`, `columns_to_select`, `replace_miss` are the same seen for the function `add_k_years_ago_statistics`.

current_year

By default, the parameter `current_year` is `False`. I.e. the current year (2020 in our case) is not considered.
Otherwise, if `current_year` is `True`, the current year is considered: that means that not only the function `add_k_years_ago_statistics` is applied two times, but also the function `add_current_year_statistics` is applied one time. The final values in the DataFrame are computed calculating the mean of the values produced by these three functions.

```
>>>df_mean_temp,_,_ = tsp.add_upTo_k_years_ago_statistics(df_mean,
df_upTo_k_years_ago=df_full_mean, k=2, current_year=True,
days_to_select=11, current_day=False, y_col="mean")
UserWarning: For the day 2020-01-01 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06']
UserWarning: For the day 2020-01-02 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07']
UserWarning: For the day 2020-01-03 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07',
'2019-01-08']
UserWarning: For the day 2020-01-04 only these 1 years ago days
have been found: ['2019-01-01', '2019-01-02', '2019-01-03',
'2019-01-04', '2019-01-05', '2019-01-06', '2019-01-07',
'2019-01-08', '2019-01-09']
...

>>> df_mean_temp
                mean  upTo_2_years_ago_mean
Datetime
2020-01-01  76.974569              24.944020
2020-01-02  56.675791              27.880914
2020-01-03  55.216906              30.439158
2020-01-04  54.887035              32.203776
2020-01-05  28.192059              32.972802
...               ...                    ...
2020-12-27  14.997987              33.063248
2020-12-28  16.317778              31.808265
2020-12-29  23.536875              30.577231
2020-12-30  22.759021              30.369128
2020-12-31  22.005000              30.312998

[366 rows x 2 columns]
```

The semantics of the parameters `current_day` is the same seen for the function `add_current_year_statistics`.