PyMove

Release 2019

Jul 20, 2021

References

PyMove is a Python library for processing and visualization of trajectories and other spatial-temporal data.

We will also release wrappers to some useful Java libraries frequently used in the mobility domain.

CHAPTER 1

Main Features

PyMove **proposes**:

- A familiar and similar syntax to Pandas;
- Clear documentation;
- Extensibility, since you can implement your main data structure by manipulating other data structures such as Dask DataFrame, numpy arrays, etc., in addition to adding new modules;
- Flexibility, as the user can switch between different data structures;
- Operations for data preprocessing, pattern mining and data visualization.

CHAPTER 2

Installation

CHAPTER 3

Conda instalation

1. `conda install -c conda-forge pymove`

CHAPTER 4

Pip installation

1. `pip install pymove`

4.1 Indices and tables

- genindex
- modindex
- search

4.1.1 pymove package

4.1.1.1 Subpackages

pymove.core package

Submodules

pymove.core.dask module

DaskMoveDataFrame class.

class pymove.core.dask.**DaskMoveDataFrame**(*data: DataFrame | list | dict*, *latitude: str = 'lat'*, *longitude: str = 'lon'*, *datetime: str = 'datetime'*, *traj_id: str = 'id'*, *n_partitions: int = 1*)

Bases: dask.dataframe.core.DataFrame, *pymove.core.interface.MoveDataFrameAbstractModel*

PyMove dataframe extending Dask DataFrame.

all (**args*, ***kwargs*)
Indicates if all elements are True, potentially over an axis.

any (**args*, ***kwargs*)
Indicates if any element is True, potentially over an axis.

append (**args*, ***kwargs*)
Append rows of other to the end of caller, returning a new object.

astype (**args*, ***kwargs*)
Casts a dask object to a specified dtype.

at
Access a single value for a row/column label pair.

columns
The column labels of the DataFrame.

convert_to (*new_type: str*) → MoveDataFrame | 'PandasMoveDataFrame' | 'DaskMoveDataFrame'
Convert an object from one type to another specified by the user.

Parameters **new_type** (*'pandas' or 'dask'*) – The type for which the object will be converted.

Returns The converted object.

Return type A subclass of MoveDataFrameAbstractModel

copy (**args*, ***kwargs*)
Make a copy of this object'srs indices and data.

count (**args*, ***kwargs*)
Counts the non-NA cells for each column or row.

datetime
Checks for the DATETIME column and returns its value.

Returns DATETIME column

Return type Series

Raises `AttributeError` – If the DATETIME column is not present in the DataFrame

describe (**args*, ***kwargs*)
Generate descriptive statistics.

drop (**args*, ***kwargs*)
Drops specified rows or columns of the dask Dataframe.

drop_duplicates (**args*, ***kwargs*)
Removes duplicated rows from the data.

dropna (**args*, ***kwargs*)
Removes missing data from dask DataFrame.

dtypes
Return the dtypes in the DataFrame.

duplicated (**args*, ***kwargs*)
Returns boolean Series denoting duplicate rows.

fillna (**args*, ***kwargs*)
Fills missing data in the dask DataFrame.

generate_date_features(**args*, ***kwargs*)
Create or update date feature.

generate_datetime_in_format_cyclical(**args*, ***kwargs*)
Create or update column with cyclical datetime feature.

generate_day_of_the_week_features(**args*, ***kwargs*)
Create or update a feature day of the week from datatime.

generate_dist_features(**args*, ***kwargs*)
Create the three distance in meters to an GPS point P.

generate_dist_time_speed_features(**args*, ***kwargs*)
Creates features of distance, time and speed between points.

generate_hour_features(**args*, ***kwargs*)
Create or update hour feature.

generate_move_and_stop_by_radius(**args*, ***kwargs*)
Create or update column with move and stop points by radius.

generate_speed_features(**args*, ***kwargs*)
Create the three speed in meters by seconds to an GPS point P.

generate_tid_based_on_id_datetime(**args*, ***kwargs*)
Create or update trajectory id based on id e datetime.

generate_time_features(**args*, ***kwargs*)
Create the three time in seconds to an GPS point P.

generate_time_of_day_features(**args*, ***kwargs*)
Create a feature time of day or period from datatime.

generate_weekend_features(**args*, ***kwargs*)
Create or update the feature weekend to the dataframe.

get_bbox(**args*, ***kwargs*)
Creates the bounding box of the trajectories.

get_type() → str
Returns the type of the object.

Returns A string representing the type of the object.

Return type str

get_users_number(**args*, ***kwargs*)
Check and return number of users in trajectory data.

groupby(**args*, ***kwargs*)
Groups dask DataFrame using a mapper or by a Series of columns.

head(*n: int = 5*, *npartitions: int = 1*, *compute: bool = True*) → dask.dataframe.core.DataFrame
Return the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters

- **n** (*int, optional, default 5*) – Number of rows to select.
- **npartitions** (*int, optional, default 1.*) – Represents the number partitions.

- **compute** (*bool, optional, default True.*) – Wether to perform the operation

Returns The first n rows of the caller object.

Return type same type as caller

iloc
Purely integer-location based indexing for selection by position.

index
The row labels of the DataFrame.

info(**args*, ***kwargs*)
Print a concise summary of a DataFrame.

isin(**args*, ***kwargs*)
Determines whether each element is contained in values.

isna(**args*, ***kwargs*)
Detect missing values.

join(**args*, ***kwargs*)
Join columns of another DataFrame.

lat
Checks for the LATITUDE column and returns its value.

Returns LATITUDE column

Return type Series

Raises AttributeError – If the LATITUDE column is not present in the DataFrame

len(**args*, ***kwargs*)
Returns the length/row numbers in trajectory data.

lng
Checks for the LONGITUDE column and returns its value.

Returns LONGITUDE column

Return type Series

Raises AttributeError – If the LONGITUDE column is not present in the DataFrame

loc
Access a group of rows and columns by label(srs) or a boolean array.

max(**args*, ***kwargs*)
Return the maximum of the values for the requested axis.

memory_usage(**args*, ***kwargs*)
Return the memory usage of each column in bytes.

merge(**args*, ***kwargs*)
Merge columns of another DataFrame.

min(**args*, ***kwargs*)
Return the minimum of the values for the requested axis.

nunique(**args*, ***kwargs*)
Count distinct observations over requested axis.

plot(**args*, ***kwargs*)
Plot the data of the dask DataFrame.

plot_all_features(**args*, ***kwargs*)
Generate a visualization for each column that type is equal dtype.

plot_traj_id(**args*, ***kwargs*)
Generate a visualization for a trajectory with the specified tid.

plot_trajs(**args*, ***kwargs*)
Generate a visualization that show trajectories.

rename(**args*, ***kwargs*)
Alter axes labels..

reset_index(**args*, ***kwargs*)
Resets the dask DataFrame'srs index, and use the default one.

sample(**args*, ***kwargs*)
Samples data from the dask DataFrame.

select_dtypes(**args*, ***kwargs*)
Returns a subset of the columns based on the column dtypes.

set_index(**args*, ***kwargs*)
Set of row labels using one or more existing columns or arrays.

shape
Return a tuple representing the dimensionality of the DataFrame.

shift(**args*, ***kwargs*)
Shifts by desired number of periods with an optional time freq.

show_trajectories_info(**args*, ***kwargs*)
Show dataset information from dataframe.

sort_values(**args*, ***kwargs*)
Sorts the values of the dask DataFrame.

tail(*n: int = 5*, *npartitions: int = 1*, *compute: bool = True*) → dask.dataframe.core.DataFrame
Return the last n rows.

This function returns the last n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters

- **n** (*int, optional, default 5*) – Number of rows to select.
- **npartitions** (*int, optional, default 1.*) – Represents the number partitions.
- **compute** (*bool, optional, default True.*) – ?

Returns The last n rows of the caller object.

Return type same type as caller

time_interval(**args*, ***kwargs*)
Get time difference between max and min datetime in trajectory.

to_csv(**args*, ***kwargs*)
Write object to a comma-separated values (csv) file.

to_data_frame() → dask.dataframe.core.DataFrame
Converts trajectory data to DataFrame format.

Returns Represents the trajectory in DataFrame format.

Return type dask.dataframe.DataFrame

to_dict(**args*, ***kwargs*)
Converts trajectory data to dict format.

to_grid(**args*, ***kwargs*)
Converts trajectory data to grid format.

to_numpy(**args*, ***kwargs*)
Converts trajectory data to numpy array format.

unique(**args*, ***kwargs*)
Return unique values of Series object.

values
Return a Numpy representation of the DataFrame.

write_file(**args*, ***kwargs*)
Write trajectory data to a new file.

pymove.core.dataframe module

MoveDataFrame class.

class pymove.core.dataframe.**MoveDataFrame**
Bases: object

Auxiliary class to check and transform data into Pymove Dataframes.

static format_labels(*current_id: str*, *current_lat: str*, *current_lon: str*, *current_datetime: str*) → dict
Format the labels for the PyMove lib pattern labels output lat, lon and datatime.

Parameters

- **current_id** (*str*) – Represents the column name of feature id
- **current_lat** (*str*) – Represents the column name of feature latitude
- **current_lon** (*str*) – Represents the column name of feature longitude
- **current_datetime** (*str*) – Represents the column name of feature datetime

Returns Represents a dict with mapping current columns of data to format of PyMove column.

Return type Dict

static has_columns(*data: pandas.core.frame.DataFrame*) → bool
Checks whether the received dataset has 'lat', 'lon', 'datetime' columns.

Parameters **data** (*DataFrame*) – Input trajectory data

Returns Represents whether or not you have the required columns

Return type bool

static validate_move_data_frame(*data: pandas.core.frame.DataFrame*)
Converts the column type to the default type used by PyMove lib.

Parameters **data** (*DataFrame*) – Input trajectory data

Raises

- KeyError – If missing one of lat, lon, datetime columns
- *ValueError, ParserError* – If the data types can't be converted

pymove.core.grid module

Grid class.

class pymove.core.grid.**Grid**(*data: DataFrame | dict, cell_size: float | None = None, meters_by_degree: float | None = None*)

Bases: object

PyMove class representing a grid.

convert_one_index_grid_to_two(*data: pandas.core.frame.DataFrame, label_grid_index: str = 'index_grid'*)

Converts grid lat-lon ids to unique values.

Parameters

- **data** (*DataFrame*) – Dataframe with grid lat-lon ids
- **label_grid_index** (*str, optional*) – grid unique id column, by default INDEX_GRID

convert_two_index_grid_to_one(*data: pandas.core.frame.DataFrame, label_grid_lat: str = 'index_grid_lat', label_grid_lon: str = 'index_grid_lon'*)

Converts grid lat-lon ids to unique values.

Parameters

- **data** (*DataFrame*) – Dataframe with grid lat-lon ids
- **label_grid_lat** (*str, optional*) – grid lat id column, by default INDEX_GRID_LAT
- **label_grid_lon** (*str, optional*) – grid lon id column, by default INDEX_GRID_LON

create_all_polygons_on_grid()

Create all polygons that are represented in a grid.

Stores the polygons in the *grid_polygon* key

create_all_polygons_to_all_point_on_grid(*data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Create all polygons to all points represented in a grid.

Parameters **data** (*DataFrame*) – Represents the dataset with contains lat, long and datetime

Returns Represents the same dataset with new key 'polygon' where polygons were saved.

Return type DataFrame

create_one_polygon_to_point_on_grid(*index_grid_lat: int, index_grid_lon: int*) → shapely.geometry.polygon.Polygon

Create one polygon to point on grid.

Parameters

- **index_grid_lat** (*int*) – Represents index of grid that reference latitude.
- **index_grid_lon** (*int*) – Represents index of grid that reference longitude.

Returns Represents a polygon of this cell in a grid.

Return type Polygon

create_update_index_grid_feature(*data: pandas.core.frame.DataFrame*, *unique_index: bool = True*, *label_dtype: Callable = <class 'numpy.int64'>*, *sort: bool = True*)

Create or update index grid feature.

It is not necessary pass dic_grid, because it creates a dic_grid if not provided.

Parameters

- **data** (*DataFrame*) – Represents the dataset with contains lat, long and datetime.
- **unique_index** (*bool, optional*) – How to index the grid, by default True
- **label_dtype** (*Callable, optional*) – Represents the type of a value of new column in dataframe, by default np.int64
- **sort** (*bool, optional*) – Represents if needs to sort the dataframe, by default True

get_grid() → dict

Returns the grid object in a dict format.

Returns

Dict with grid information 'lon_min_x': minimum x of grid, 'lat_min_y': minimum y of grid, 'grid_size_lat_y': lat y size of grid, 'grid_size_lon_x': lon x size of grid, 'cell_size_by_degree': cell size in radians

Return type Dict

point_to_index_grid(*event_lat: float*, *event_lon: float*) → tuple[int, int]

Locate the coordinates x and y in a grid of point (lat, long).

Parameters

- **event_lat** (*float*) – Represents the latitude of a point
- **event_lon** (*float*) – Represents the longitude of a point

Returns Represents the index y in a grid of a point (lat, long) Represents the index x in a grid of a point (lat, long)

Return type Tuple[int, int]

read_grid_pkl(*filename: str*) → Grid

Read grid dict from a file .pkl.

Parameters **filename** (*str*) – Represents the name of a file.

Returns Grid object containing informations about virtual grid

Return type *Grid*

save_grid_pkl(*filename: str*)

Save a grid with new file .pkl.

Parameters **filename** (*Text*) – Represents the name of a file.

pymove.core.interface module

class pymove.core.interface.**MoveDataFrameAbstractModel**

Bases: abc.ABC

all()

any()

append()

astype()

at()

columns()

convert_to(*new_type: str*)

copy()

count()

datetime()

describe()

drop()

drop_duplicates()

dropna()

dtypes()

duplicated()

fillna()

generate_date_features()

generate_datetime_in_format_cyclical()

generate_day_of_the_week_features()

generate_dist_features()

generate_dist_time_speed_features()

generate_hour_features()

generate_move_and_stop_by_radius()

generate_speed_features()

generate_tid_based_on_id_datetime()

generate_time_features()

generate_time_of_day_features()

generate_weekend_features()

get_bbox()

get_type()

get_users_number()

groupby()

head()

iloc()

index()

info()

isin()

isna()

join()

lat()

len()

lng()

loc()

max()

memory_usage()

merge()

min()

nunique()

plot()

plot_all_features()

plot_traj_id()

plot_trajs()

rename()

reset_index()

sample()

select_dtypes()

set_index()

shape()

shift()

show_trajectories_info()

sort_values()

tail()

time_interval()

to_csv()

to_data_frame()

to_dict()

to_grid()

to_numpy()

values()

write_file()

pymove.core.pandas module

PandasMoveDataFrame class.

class pymove.core.pandas.**PandasMoveDataFrame**(*data: DataFrame | list | dict*, *latitude: str = 'lat'*, *longitude: str = 'lon'*, *datetime: str = 'datetime'*, *traj_id: str = 'id'*)

Bases: pandas.core.frame.DataFrame

PyMove dataframe extending Pandas DataFrame.

append(*other: 'PandasMoveDataFrame' | DataFrame*, *ignore_index: bool = False*, *verify_integrity: bool = False*, *sort: bool = False*) → 'PandasMoveDataFrame'

Append rows of other to the end of caller, returning a new object.

Columns in other that are not in the caller are added as new columns.

Parameters

- **other** (*DataFrame or Series/dict-like object, or list of these*) – The data to append.
- **ignore_index** (*bool, optional*) – If True, do not use the index labels, by default False
- **verify_integrity** (*bool, optional*) – If True, raise ValueError on creating index with duplicates, by default False
- **sort** (*bool, optional*) – Sort columns if the columns of self and other are not aligned The default sorting is deprecated and will change to not-sorting in a future version of pandas. by default False

Returns A dataframe containing rows from both the caller and *other*.

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html

astype(*dtype: Callable | dict*, *copy: bool = True*, *errors: str = 'raise'*) → DataFrame

Cast a pandas object to a specified dtype.

Parameters

- **dtype** (*callable, dict*) – Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame columns to column-specific types.
- **copy** (*bool, optional*) – Return a copy when copy=True (be very careful setting copy=False as changes to values then may propagate to other pandas objects), by default True
- **errors** (*str, optional*) –

  Control raising of exceptions on invalid data for provided dtype, by default 'raise

  – raise : allow exceptions to be raised

  – ignore : suppress exceptions. On error return original object

Returns Casted object to specified type.

Return type DataFrame

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html

Raises `AttributeError` – If trying to change required types inplace

convert_to(*new_type: str*) → MoveDataFrame | 'PandasMoveDataFrame' | 'DaskMoveDataFrame'
Convert an object from one type to another specified by the user.

Parameters **new_type** (*'pandas' or 'dask'*) – The type for which the object will be converted.

Returns The converted object.

Return type A subclass of MoveDataFrameAbstractModel

copy(*deep: bool = True*) → PandasMoveDataFrame
Make a copy of this object's indices and data.

When deep=True (default), a new object will be created with a copy of the calling object data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below). When deep=False, a new object will be created without copying the calling object data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters **deep** (*bool, optional*) – Make a deep copy, including a copy of the data and the indices. With deep=False neither the indices nor the data are copied, by default True

Returns Object type matches caller.

Return type *PandasMoveDataFrame*

Notes

When deep=True, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to copy.deepcopy in the Standard Library, which recursively copies object data (see examples below). While Index objects are copied when deep=True, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.

datetime
Checks for the DATETIME column and returns its value.

Returns DATETIME column

Return type Series

Raises `AttributeError` – If the DATETIME column is not present in the DataFrame

drop(*labels: str | list[str] | None = None*, *axis: int | str = 0*, *index: str | list[str] | None = None*, *columns: str | list[str] | None = None*, *level: int | str | None = None*, *inplace: bool = False*, *errors: str = 'raise'*) → 'PandasMoveDataFrame' | DataFrame | None
Removes rows or columns.

By specifying label names and corresponding axis, or by specifying directly index or column names. When using a multiindex, labels on different levels can be removed by specifying the level.

Parameters

- **labels** (*str or list, optional*) – Index or column labels to drop, by default None
- **axis** (*int or str, optional*) – Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns'), by default 0
- **index** (*str or list, optional*) – Alternative to specifying axis (labels, axis=0 is equivalent to index=labels), by default None
- **columns** (*str or list, optional*) – Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels), by default None
- **level** (*str or int, optional*) – For MultiIndex, level from which the labels will be removed, by default None
- **inplace** (*bool, optional*) – If True, do operation inplace and return None Otherwise, make a copy, do operations and return, by default False
- **errors** (*bool, optional*) – 'ignore', 'raise', by default 'raise' If 'ignore', suppress error and only existing labels are dropped.

Returns Object without the removed index or column labels or None

Return type *PandasMoveDataFrame*, DataFrame

Raises

- `AttributeError` – If trying to drop a required column inplace
- `KeyError` – If any of the labels is not found in the selected axis.

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

drop_duplicates(*subset: int | str | None = None*, *keep: str | bool = 'first'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | None

Uses the pandas's function drop_duplicates, to remove duplicated rows from data.

Parameters

- **subset** (*int or str, optional*) – Only consider certain columns for identifying duplicates, by default use all of the columns, by default None
- **keep** (*str, optional*) –
  - first : Drop duplicates except for the first occurrence.
  - last : Drop duplicates except for the last occurrence.
  - False : Drop all duplicates.

  by default 'first'
- **inplace** (*bool, optional*) – Whether to drop duplicates in place or to return a copy, by default False

Returns Object with duplicated rows or None

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html

dropna(*axis: int | str = 0*, *how: str = 'any'*, *thresh: float | None = None*, *subset: list | None = None*, *inplace: bool = False*)
Removes missing data.

Parameters

- **axis** (*0 or 'index', 1 or 'columns', None, optional*) – Determine if rows or columns are removed, by default 0 - 0, or 'index' : Drop rows which contain missing values. - 1, or 'columns' : Drop columns which contain missing value.
- **how** (*str, optional*) – Determine if row or column is removed from DataFrame, by default 'any when we have at least one NA or all NA.
  - 'any' : If any NA values are present, drop that row or column.
  - 'all' : If all values are NA, drop that row or column.
- **thresh** (*float, optional*) – Require that many non-NA values, by default None
- **subset** (*array-like, optional*) – Labels along other axis to consider, by default None e.g. if you are dropping rows these would be a list of columns to include.
- **inplace** (*bool, optional*) – If True, do operation inplace and return None, by default False

Returns Object with NA entries dropped or None

Return type *PandasMoveDataFrame*

References

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

Raises `AttributeError` – If trying to drop required columns inplace

fillna(*value: Any | None = None*, *method: str | None = None*, *axis: int | str | None = None*, *inplace: bool = False*, *limit: int | None = None*, *downcast: dict | None = None*)
Fill NA/NaN values using the specified method.

Parameters

- **value** (*scalar, dict, Series, or DataFrame*) – Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.
- **method** (*{'backfill', 'bfill', 'pad', 'ffill', None}, default None*) – Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Axis along which to fill missing values.
- **inplace** (*bool, default False*) – If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

- **limit** (*int, default None*) – If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.
- **downcast** (*dict, default is None*) – A dict of item->dtype of what to downcast if possible, or the str 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

Returns Object with missing values filled or None

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html

generate_date_features(*inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create or update date feature based on datetime.

Parameters **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

generate_datetime_in_format_cyclical(*label_datetime: str = 'datetime', inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create or update column with cyclical datetime feature.

Parameters

- **label_datetime** (*str, optional*) – Represents column id type, by default DATETIME
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

References

https://ianlondon.github.io/blog/encoding-cyclical-features-24hour-time/ https://www.avanwyk.com/encoding-cyclical-features-for-deep-learning/

generate_day_of_the_week_features(*inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create or update day of the week features based on datetime.

Parameters **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

generate_dist_features(*label_id: str = 'id'*, *label_dtype: Callable = <class 'numpy.float64'>*, *sort: bool = True*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create the three distance in meters to an GPS point P.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectories id, by default TRAJ_ID
- **label_dtype** (*callable, optional*) – Represents column id type, by default np.float64
- **sort** (*bool, optional*) – If sort == True the dataframe will be sorted, by True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

Examples

- P to P.next = 2 meters
- P to P.previous = 1 meter
- P.previous to P.next = 1 meters

generate_dist_time_speed_features(*label_id: str = 'id'*, *label_dtype: Callable = <class 'numpy.float64'>*, *sort: bool = True*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None

Adds distance, time and speed information to the dataframe.

Firstly, create the three distance to an GPS point P (lat, lon). After, create two time features to point P: time to previous and time to next. Lastly, create two features to speed using time and distance features.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectories id, by default TRAJ_ID
- **label_dtype** (*callable, optional*) – Represents column id type, by default np.float64
- **sort** (*bool, optional*) – If sort == True the dataframe will be sorted, by True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

Examples

- dist_to_prev = 248.33 meters, dist_to_prev 536.57 meters
- time_to_prev = 60 seconds, time_prev = 60.0 seconds
- speed_to_prev = 4.13 m/srs, speed_prev = 8.94 m/srs.

generate_hour_features(*inplace: bool = True*) → 'PandasMoveDataFrame' | None
Create or update hour features based on datetime.

Parameters **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

generate_move_and_stop_by_radius(*radius: float = 0*, *target_label: str = 'dist_to_prev'*, *inplace: bool = True*)
Create or update column with move and stop points by radius.

Parameters

- **radius** (*float, optional*) – Represents radius, by default 0
- **target_label** (*str, optional*) – Represents column to compute, by default DIST_TO_PREV
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

generate_speed_features(*label_id: str = 'id'*, *label_dtype: Callable = <class 'numpy.float64'>*, *sort: bool = True*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None
Create the three speed in meter by seconds to an GPS point P.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectories id, by default TRAJ_ID
- **label_dtype** (*callable, optional*) – Represents column id type, by default np.float64
- **sort** (*bool, optional*) – If sort == True the dataframe will be sorted, by True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

Raises `ValueError` – If feature generation fails

Examples

- P to P.next = 1 meter/seconds
- P to P.previous = 3 meter/seconds
- P.previous to P.next = 2 meter/seconds

generate_tid_based_on_id_datetime(*str_format: str = '%Y%m%d%H'*, *sort: bool = True*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None
Create or update trajectory id based on id and datetime.

Parameters

- **str_format** (*str, optional*) – Format to consider the datetime, by default '%Y%m%d%H'
- **sort** (*bool, optional*) – Wether to sort the dataframe, by default True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

generate_time_features(*label_id: str = 'id'*, *label_dtype: Callable = <class 'numpy.float64'>*, *sort: bool = True*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create the three time in seconds to an GPS point P.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectories id, by default TRAJ_ID
- **label_dtype** (*callable, optional*) – Represents column id type, by default np.float64
- **sort** (*bool, optional*) – If sort == True the dataframe will be sorted, by True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

Examples

- P to P.next = 5 seconds
- P to P.previous = 15 seconds
- P.previous to P.next = 20 seconds

generate_time_of_day_features(*inplace: bool = True*) → 'PandasMoveDataFrame' | None

Create or update time of day features based on datetime.

Parameters **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns

Object with new features or None Early morning from 0H to 6H Morning from 6H to 12H Afternoon from 12H to 18H Evening from 18H to 24H

Return type *PandasMoveDataFrame*

Examples

- datetime1 = 2019-04-28 02:00:56 -> period = Early Morning
- datetime2 = 2019-04-28 08:00:56 -> period = Morning

- datetime3 = 2019-04-28 14:00:56 -> period = Afternoon
- datetime4 = 2019-04-28 20:00:56 -> period = Evening

generate_weekend_features(*create_day_of_week: bool = False*, *inplace: bool = True*) → 'PandasMoveDataFrame' | None

Adds information to rows determining if it is a weekend day.

Create or update the feature weekend to the dataframe, if this resource indicates that the given day is the weekend, otherwise, it is a day of the week.

Parameters

- **create_day_of_week** (*bool, optional*) – Indicates if the column day should be keeped in the dataframe. If set to False the column will be dropped, by default False
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasMoveDataFrame*

get_bbox() → tuple[float, float, float, float]

Returns the bounding box of the dataframe.

A bounding box (usually shortened to bbox) is an area defined by two longitudes and two latitudes, where:

- Latitude is a decimal number between -90.0 and 90.0.
- Longitude is a decimal number between -180.0 and 180.0.

They usually follow the standard format of: - bbox = left, bottom, right, top - bbox = min Longitude , min Latitude , max Longitude , max Latitude

Returns Represents a bound box, that is a tuple of 4 values with the min and max limits of latitude e longitude. lat_min, lon_min, lat_max, lon_max

Return type Tuple[float, float, float, float]

Examples

(22.147577, 113.54884299999999, 41.132062, 121.156224)

get_type() → str

Returns the type of the object.

Returns A string representing the type of the object.

Return type str

get_users_number() → int

Check and return number of users in trajectory data.

Returns Represents the number of users in trajectory data.

Return type int

head(*n: int = 5*) → PandasMoveDataFrame

Return the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters **n** (*int, optional*) – Number of rows to select, by default 5

Returns The first n rows of the caller object.

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.head.html

isin(*values: list | Series | DataFrame | dict*) → DataFrame
Determines whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If values is a Series, the index. If values is a dict, the keys must be the column names, which must match. If values is a DataFrame, then both the index and column labels must match.

Returns DataFrame of booleans showing whether each element in the DataFrame is contained in values

Return type DataFrame

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isin.html

join(*other: 'PandasMoveDataFrame' | DataFrame*, *on: str | list | None = None*, *how: str = 'left'*, *lsuffix: str = ''*, *rsuffix: str = ''*, *sort: bool = False*) → 'PandasMoveDataFrame'
Join columns of other, returning a new object.

Join columns with *other* PandasMoveDataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

- **other** (*DataFrame, Series, or list of DataFrame*) – Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.
- **on** (*str or list of str or array-like, optional*) – Column or index level name(srs) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.
- **how** (*{'left', 'right', 'outer', 'inner'}, optional*) – How to handle the operation of the two objects, by default 'left'
  - left: use calling frame index (or column if on is specified)
  - right: use *other* index.
  - outer: form union of calling frame index (or column if on is

  specified) with *other* index, and sort it. lexicographically. * inner: form intersection of calling frame index (or column if on is specified) with *other* index, preserving the order of the calling one.
- **lsuffix** (*str, optional*) – Suffix to use from left frame overlapping columns, by default ''
- **rsuffix** (*str, optional*) – Suffix to use from right frame overlapping columns, by default ''

- **sort** (*bool, optional*) – Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

Returns A dataframe containing columns from both the caller and *other*.

Return type *PandasMoveDataFrame*

Notes

Parameters *on*, *lsuffix*, and *rsuffix* are not supported when passing a list of *DataFrame* objects.

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.join.html

lat

Checks for the LATITUDE column and returns its value.

Returns LATITUDE column

Return type Series

Raises AttributeError – If the LATITUDE column is not present in the DataFrame

len() → int

Returns the length/row numbers in trajectory data.

Returns Represents the trajectory data length.

Return type int

lng

Checks for the LONGITUDE column and returns its value.

Returns LONGITUDE column

Return type Series

Raises AttributeError – If the LONGITUDE column is not present in the DataFrame

merge(*right: 'PandasMoveDataFrame' | DataFrame | Series*, *how: str = 'inner'*, *on: str | list | None = None*, *left_on: str | list | None = None*, *right_on: str | list | None = None*, *left_index: bool = False*, *right_index: bool = False*, *sort: bool = False*, *suffixes: tuple[str, str] = ('_x', '_y')*, *copy: bool = True*, *indicator: bool | str = False*, *validate: str | None = None*) → 'PandasMoveDataFrame'

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

- **right** (*DataFrame or named Series*) – Object to merge with.
- **how** (*{'left', 'right', 'outer', 'inner'}, optional*) – Type of merge to be performed, by default 'inner' left: use only keys from left frame, similar to a SQL left outer join;

  preserve key order.

  right: use only keys from right frame, similar to a SQL right outer join; preserve key order.

outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.

inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

- **on** (*label or list, optional*) – Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames, by default None
- **left_on** (*str or list or array-like, optional*) – Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns, by default None
- **right_on** (*str or list or array-like, optional*) – Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns, by default None
- **left_index** (*bool, optional*) – Use the index from the left DataFrame as the join key(s), by default False If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.
- **right_index** (*bool, optional*) – Use the index from the right DataFrame as the join key, by default False Same caveats as left_index.
- **sort** (*bool, optional*) – Sort the join keys lexicographically in the result DataFrame, by default False If False, the order of the join keys depends on the join type (how keyword).
- **suffixes** (*tuple of (str, str), optional*) – Suffix to apply to overlapping column names in the left and right side respectively. To raise an exception on overlapping columns use (False, False) by default ('_x', '_y')
- **copy** (*bool, optional*) – If False, avoid copy if possible, by default True
- **indicator** (*bool or str, optional*) – If True, adds a column to output DataFrame called '_merge' with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of 'left_only' for observations whose merge key only appears in 'left' DataFrame, 'right_only' for observations whose merge key only appears in 'right' DataFrame, and 'both' if the observation's merge key is found in both. by default False
- **validate** (*str, optional*) – If specified, checks if merge is of specified type, by default None 'one_to_one' or '1:1': check if merge keys are unique in both

  left and right datasets.

  'one_to_many' or '1:m': check if merge keys are unique in left dataset. 'many_to_one' or 'm:1': check if merge keys are unique in right dataset. 'many_to_many' or 'm:m': allowed, but does not result in checks.

Returns A DataFrame of the two merged objects.

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html?highlight=merge#pandas.DataFrame.merge

rename(*mapper: dict | Callable | None = None*, *index: dict | Callable | None = None*, *columns: dict | Callable | None = None*, *axis: int | str | None = None*, *copy: bool = True*, *inplace: bool = False*) → 'PandasMoveDataFrame' | DataFrame | None

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Parameters

- **mapper** (*dict or function, optional*) – Dict-like or functions transformations to apply to that axis' values. Use either mapper and axis to specify the axis to target with mapper, or index and columns, by default None
- **index** (*dict or function, optional*) – Alternative to specifying axis (mapper, axis=0 is equivalent to index=mapper), by default None
- **columns** (*dict or function, optional*) – Alternative to specifying axis (mapper, axis=1 is equivalent to columns=mapper), by default None
- **axis** (*int or str, optional*) – Axis to target with mapper. Can be either the axis name ('index', 'columns') or number (0, 1), by default None
- **copy** (*bool, optional*) – Also copy underlying data, by default True
- **inplace** (*bool, optional*) – Whether to return a new DataFrame. If True then value of copy is ignored, by default False

Returns DataFrame with the renamed axis labels or None

Return type *PandasMoveDataFrame*, DataFrame

Raises AttributeError – If trying to rename a required column inplace

reset_index(*level: int | str | tuple | list | None = None*, *drop: bool = False*, *inplace: bool = False*, *col_level: int | str = 0*, *col_fill: str = ''*) → 'PandasMoveDataFrame' | None

Resets the DataFrame's index, and use the default one.

One or more levels can be removed, if the DataFrame has a MultiIndex.

Parameters

- **level** (*int or str or tuple or list, optional*) – Only the levels specify will be removed from the index If set to None, all levels are removed, by default None
- **drop** (*bool, optional*) – Do not try to insert index into dataframe columns This resets the index to the default integer index, by default False
- **inplace** (*bool, optional*) – Modify the DataFrame in place (do not create a new object), by default False
- **col_level** (*int or str, optional*) – If the columns have multiple levels, determines which level the labels are inserted into, by default 0
- **col_fill** (*str, optional*) – If the columns have multiple levels, determines how the other levels are named If None then the index name is repeated, by default ''
- **PandasMoveDataFrame** – The generated picture or None

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html

sample(*n: int | None = None*, *frac: float | None = None*, *replace: bool = False*, *weights: str | list | None = None*, *random_state: int | None = None*, *axis: int | str | None = None*) → 'PandasMoveDataFrame'

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

- **n** (*int, optional*) – Number of items from axis to return. Cannot be used with *frac*, by default None
- **frac** (*float, optional*) – Fraction of axis items to return. Cannot be used with *n*, by deault None
- **replace** (*bool, optional*) – Allow or disallow sampling of the same row more than once, by default False
- **weights** (*str or ndarray-like, optional*) – If 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed. by default None
- **random_state** (*int or numpy.random.RandomState, optional*) – Seed for the random number generator (if int), or numpy RandomState object,by default None
- **axis** (*{0 or 'index', 1 or 'columns', None}, optional*) – Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames), by default None

Returns A new object of same type as caller containing *n* items randomly sampled from the caller object.

Return type *PandasMoveDataFrame*

See also:

numpy.random.choice() Generates a random sample from a given 1-D numpy array.

Notes

If *frac* > 1, *replacement* should be set to *True*.

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sample.html

set_index(*keys: str | list[str], drop: bool = True, append: bool = False, inplace: bool = False, verify_integrity: bool = False*) → 'PandasMoveDataFrame' | DataFrame | None

Set the DataFrame index (row labels) using one or more existing columns or arrays.

Parameters

- **keys** (*str, list*) – label or array-like or list of labels/arrays This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays
- **drop** (*bool, optional*) – Delete columns to be used as the new index, by default True
- **append** (*bool, optional*) – Whether to append columns to existing index, by default True
- **inplace** (*bool, optional*) – Modify the DataFrame in place (do not create a new object), by default True
- **verify_integrity** (*bool, optional*) – Check the new index for duplicates Otherwise defer the check until necessary Setting to False will improve the performance of this method, by default True

Returns Object with a new index or None

Return type *PandasMoveDataFrame*, DataFrame

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html

Raises `AttributeError` – If trying to change required columns types

shift (*periods: int = 1, freq: DateOffset | Timedelta | str | None = None, axis: int | str = 0, fill_value: Any | None = None*) → 'PandasMoveDataFrame'

Shift index by desired number of periods with an optional time freq.

Parameters

- **periods** (*int, optional, default 1*) – Number of periods to shift. Can be positive or negative.
- **freq** (*DateOffset or Timedelta or str, optional, default None*) – Offset to use from the series module or time rule (e.g. 'EOM'). If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data. When freq is not passed, shift the index without realigning the data. If freq is passed (in this case, the index must be date or datetime, or it will raise a NotImplementedError), the index will be increased using the periods and the freq.
- **axis** (*0 or 'index', 1 or 'columns', None, optional, default 0*) – Shift direction.
- **fill_value** (*object, optional, default None*) – The scalar value to use for newly introduced missing values. The default depends on the dtype of self. For numeric data, np.nan is used. For datetime, timedelta, or period data, etc. NaT is used. For extension dtypes, self.dtype.na_value is used.

Returns A copy of the original object, shifted.

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.shift.html

show_trajectories_info()
Show dataset information from dataframe.

Displays the number of rows, datetime interval, and bounding box.

Examples

========== INFORMATION ABOUT DATASET ========== Number of Points: 217654 Number of IDs objects: 2 Start Date:2008-10-23 05:53:05 End Date:2009-03-19 05:46:37 Bounding Box:(22.147577, 113.54884299999999, 41.132062, 121.156224) ==========

sort_values(*by: str | list[str], axis: int = 0, ascending: bool = True, inplace: bool = False, kind: str = 'quicksort', na_position: str = 'last'*) → 'PandasMoveDataFrame' | None
Sorts the values of the _data, along an axis.

Parameters

- **by** (*str, list*) – Name or list of names to sort the _data by
- **axis** (*int, optional*) – if set to 0 or 'index', will count for each column. if set to 1 or 'columns', will count for each row by default 0
- **ascending** (*bool, optional*) – Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bool, must match the length, by default True
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered, the duplicates will be dropped in place, otherwise the operation will be made in a copy, that will be returned, by default False
- **kind** (*str, optional*) – Choice of sorting algorithm, 'quicksort', 'mergesort', 'heapsort' For DataFrames, this option is only applied when sorting on a single column or label, by default 'quicksort'
- **na_position** (*str, optional*) – 'first', 'last', by default 'last If 'first' puts NaNs at the beginning; If last puts NaNs at the end.

Returns The sorted dataframe or None

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html

tail(*n: int = 5*) → PandasMoveDataFrame
Return the last n rows.

This function returns the last n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters **n** (*int, optional*) – Number of rows to select, by default 5

Returns The last n rows of the caller object.

Return type *PandasMoveDataFrame*

References

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.tail.html

time_interval() → pandas._libs.tslibs.timedeltas.Timedelta
Get time difference between max and min datetime in trajectory data.

Returns Represents the time difference.

Return type Timedelta

to_data_frame() → pandas.core.frame.DataFrame
Converts trajectory data to DataFrame format.

Returns Represents the trajectory in DataFrame format.

Return type DataFrame

to_dicrete_move_df(*local_label: str = 'local_label'*) → PandasMoveDataFrame
Generate a discrete dataframe move.

Parameters **local_label** (*str, optional*) – Represents the column name of feature local label, default LOCAL_LABEL

Returns Represents an PandasMoveDataFrame discretized.

Return type *PandasDiscreteMoveDataFrame*

to_grid(*cell_size: float*, *meters_by_degree: float | None = None*) → Grid
Converts trajectory data to grid format.

Parameters

- **cell_size** (*float*) – Represents grid cell size.
- **meters_by_degree** (*float, optional*) – Represents the corresponding meters of lat by degree, by default lat_meters(-3.71839)

Returns Represents the trajectory in grid format

Return type *Grid*

write_file(*file_name: str*, *separator: str = ','*)
Write trajectory data to a new file.

Parameters

- **file_name** (*str*) – Represents the filename.
- **separator** (*str, optional*) – Represents the information separator in a new file, by default ‘,’

pymove.core.pandas_discrete module

PandasDiscreteMoveDataFrame class.

class pymove.core.pandas_discrete.**PandasDiscreteMoveDataFrame**(*data: DataFrame | list | dict, latitude: str = 'lat', longitude: str = 'lon', datetime: str = 'datetime', traj_id: str = 'id', local_label: str = 'local_label'*)

Bases: *pymove.core.pandas.PandasMoveDataFrame*

PyMove discrete dataframe extending PandasMoveDataFrame.

discretize_based_grid(*region_size: int = 1000*)

Discrete space in cells of the same size, assigning a unique id to each cell.

Parameters **region_size** (*int, optional*) – Size of grid cell, by default 1000

generate_prev_local_features(*label_id: str = 'id', local_label: str = 'local_label', sort: bool = True, inplace: bool = True*) → 'PandasDiscreteMoveDataFrame' | None

Create a feature prev_local with the label of previous local to current point.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectory id, by default TRAJ_ID
- **local_label** (*str, optional*) –

  Indicates name of column of place labels on symbolic trajectory, by default LOCAL_LABEL
- **sort** (*bool, optional*) – Wether the dataframe will be sorted, by default True
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasDiscreteMoveDataFrame*

generate_tid_based_statistics(*label_id: str = 'id', local_label: str = 'local_label', mean_coef: float = 1.0, std_coef: float = 1.0, statistics: DataFrame | None = None, label_tid_stat: str = 'tid_stat', drop_single_points: bool = False, inplace: bool = True*) → 'PandasDiscreteMoveDataFrame' | None

Splits the trajectories into segments based on time statistics for segments.

Parameters

- **label_id** (*str, optional*) – Represents name of column of trajectory id, by default TRAJ_ID
- **local_label** (*str, optional*) –

  Indicates name of column of place labels on symbolic trajectory, by default LOCAL_LABEL
- **mean_coef** (*float, optional*) – Multiplication coefficient of the mean time for the segment, by default 1.0
- **std_coef** (*float, optional*) – Multiplication coefficient of sdt time for the segment, by default 1.0

- **statistics** (*DataFrame, optional*) – Time Statistics of the pairwise local labels, by default None
- **label_tid_stat** (*str, optional*) – The label of the column containing the ids of the formed segments. Is the new splitted id, by default TID_STAT
- **drop_single_points** (*bool, optional*) – Wether to drop the trajectories with only one point, by default False
- **inplace** (*bool, optional*) – Represents whether the operation will be performed on the data provided or in a copy, by default True

Returns Object with new features or None

Return type *PandasDiscreteMoveDataFrame*

Raises

- `KeyError` – If missing local_label column
- `ValueError` – If the data contains only null values

Module contents

Contains the core of PyMove.

MoveDataFrame, PandasMoveDataFrame, DaskMoveDataFrame, PandasDiscreteMoveDataFrame, Grid

class pymove.core.**MoveDataFrameAbstractModel**

Bases: `abc.ABC`

all()

any()

append()

astype()

at()

columns()

convert_to(*new_type: str*)

copy()

count()

datetime()

describe()

drop()

drop_duplicates()

dropna()

dtypes()

duplicated()

fillna()

generate_date_features()

generate_datetime_in_format_cyclical()

generate_day_of_the_week_features()

generate_dist_features()

generate_dist_time_speed_features()

generate_hour_features()

generate_move_and_stop_by_radius()

generate_speed_features()

generate_tid_based_on_id_datetime()

generate_time_features()

generate_time_of_day_features()

generate_weekend_features()

get_bbox()

get_type()

get_users_number()

groupby()

head()

iloc()

index()

info()

isin()

isna()

join()

lat()

len()

lng()

loc()

max()

memory_usage()

merge()

min()

nunique()

plot()

plot_all_features()

plot_traj_id()

plot_trajs()

rename()

reset_index()

sample()

select_dtypes()

set_index()

shape()

shift()

show_trajectories_info()

sort_values()

tail()

time_interval()

to_csv()

to_data_frame()

to_dict()

to_grid()

to_numpy()

values()

write_file()

pymove.models package

Subpackages

pymove.models.pattern_mining package

Submodules

pymove.models.pattern_mining.clustering module

Clustering operations.

elbow_method, gap_statistic, dbscan_clustering

pymove.models.pattern_mining.clustering.**dbscan_clustering**(*move_data: DataFrame*, *cluster_by: str*, *meters: int = 10*, *min_sample: float = 840.0*, *earth_radius: float = 6371*, *metric: str | Callable = 'euclidean'*, *inplace: bool = False*) → DataFrame | None

Performs density based clustering on the move_dataframe according to cluster_by.

Parameters

- **move_data** (*dataframe*) – the input trajectory
- **cluster_by** (*str*) – the colum to cluster
- **meters** (*int, optional*) – distance to use in the clustering, by default 10
- **min_sample** (*float, optional*) – the minimum number of samples to consider a cluster, by default 1680/2
- **earth_radius** (*int*) – Y offset from your original position in meters, by default EARTH_RADIUS
- **metric** (*string, or callable, optional*) – The metric to use when calculating distance between instances in a feature array by default 'euclidean'
- **inplace** (*bool, optional*) – Whether to return a new DataFrame, by default False

Returns Clustered dataframe or None

Return type DataFrame

pymove.models.pattern_mining.clustering.**elbow_method**(*move_data: DataFrame, k_initial: int = 1, max_clusters: int = 15, k_iteration: int = 1, random_state: int | None = None*) → dict

Determines the optimal number of clusters.

In the range set by the user using the elbow method.

Parameters

- **move_data** (*dataframe*) – The input trajectory data.
- **k_initial** (*int, optional*) – The initial value used in the interaction of the elbow method. Represents the maximum numbers of clusters, by default 1
- **max_clusters** (*int, optional*) – The maximum value used in the interaction of the elbow method. Maximum number of clusters to test for, by default 15
- **k_iteration** (*int, optional*) – Increment value of the sequence used by the elbow method, by default 1
- **random_state** (*int, RandomState instance*) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic, by default None

Returns The inertia values for the different numbers of clusters

Return type dict

Example

clustering.elbow_method(move_data=move_df, k_iteration=3)

{ 1: 55084.15957839036, 4: 245.68365592382938, 7: 92.31472644640075, 10: 62.618599956870355, 13: 45.59653757292055,

}

pymove.models.pattern_mining.clustering.**gap_statistic**(*move_data: DataFrame*, *nrefs: int = 3*, *k_initial: int = 1*, *max_clusters: int = 15*, *k_iteration: int = 1*, *random_state: int | None = None*) → dict

Calculates optimal clusters numbers using Gap Statistic.

From Tibshirani, Walther, Hastie.

Parameters

- **move_data** (*ndarray of shape (n_samples, n_features)*) – The input trajectory data.
- **nrefs** (*int, optional*) – number of sample reference datasets to create, by default 3
- **k_initial** (*int, optional.*) – The initial value used in the interaction of the elbow method, by default 1 Represents the maximum numbers of clusters.
- **max_clusters** (*int, optional*) – Maximum number of clusters to test for, by default 15
- **k_iteration** (*int, optional*) – Increment value of the sequence used by the elbow method, by default 1
- **random_state** (*int, RandomState instance*) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic, by default None

Returns The error value for each cluster number

Return type dict

Notes

https://anaconda.org/milesgranger/gap-statistic/notebook

pymove.models.pattern_mining.freq_seq_patterns module

Not implemented.

pymove.models.pattern_mining.moving_together_patterns module

Not implemented.

pymove.models.pattern_mining.periodic_patterns module

Not implemented.

Module contents

Contains models to detect patterns on trajectories.

clustering, freq_seq_patterns, moving_together_patterns, periodic_patterns

Submodules

pymove.models.anomaly_detection module

Not implemented.

pymove.models.classification module

Not implemented.

Module contents

Contains models to perform operations on trajectories.

pattern_mining, anomaly_detection, classification

pymove.preprocessing package

Submodules

pymove.preprocessing.compression module

Compression operations.

compress_segment_stop_to_point

pymove.preprocessing.compression.**compress_segment_stop_to_point**(*move_data: pandas.core.frame.DataFrame*, *label_segment: str = 'segment_stop'*, *label_stop: str = 'stop'*, *point_mean: str = 'default'*, *drop_moves: bool = False*, *label_id: str = 'id'*, *dist_radius: float = 30*, *time_radius: float = 900*, *inplace: bool = False*) → pandas.core.frame.DataFrame

Compress the trajectories using the stop points in the dataframe.

Compress a segment to point setting lat_mean e lon_mean to each segment.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_segment** (*String, optional*) – The label of the column containing the ids of the formed segments. Is the new splitted id, by default SEGMENT_STOP
- **label_stop** (*String, optional*) – Is the name of the column that indicates if a point is a stop, by default STOP
- **point_mean** (*String, optional*) – Indicates whether the mean points should be calculated using centroids or the point that repeat the most, by default 'default'
- **drop_moves** (*Boolean, optional*) – If set to true, the moving points will be dropped from the dataframe, by default False
- **label_id** (*String, optional*) – Used to create the stay points used in the compression. If the dataset already has the stop move, this parameter should be ignored. Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **dist_radius** (*Double, optional*) – Used to create the stay points used in the compression, by default 30 If the dataset already has the stop move, this parameter should be ignored. The first step in this function is segmenting the trajectory. The segments are used to find the stop points. The dist_radius defines the distance used in the segmentation.
- **time_radius** (*Double, optional*) – Used to create the stay points used in the compression, by default 900 If the dataset already has the stop move, this

  parameter should be ignored.

  The time_radius used to determine if a segment is a stop. If the user stayed in the segment for a time greater than time_radius, than the segment is a stop.
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns

Data with 3 additional features: segment_stop, lat_mean and lon_mean or None segment_stop indicates the trajectory segment to which the point belongs lat_mean and lon_mean:

if the default option is used, lat_mean and lon_mean are defined based on point that repeats most within the segment On the other hand, if centroid option is used, lat_mean and lon_mean are defined by centroid of the all points into segment

Return type DataFrame

pymove.preprocessing.filters module

Filtering operations.

get_bbox_by_radius, by_bbox, by_datetime, by_label, by_id, by_tid, clean_consecutive_duplicates, clean_gps_jumps_by_distance, clean_gps_nearby_points_by_distances, clean_gps_nearby_points_by_speed, clean_gps_speed_max_radius, clean_trajectories_with_few_points, clean_trajectories_short_and_few_points, clean_id_by_time_max

pymove.preprocessing.filters.**by_bbox**(*move_data: DataFrame, bbox: tuple[float, float, float, float], filter_out: bool = False, inplace: bool = False*) → DataFrame | None

Filters points of the trajectories according to specified bounding box.

Parameters

- **move_data** (*dataframe*) – The input trajectories data

- **bbox** (*tuple*) – Tuple of 4 elements, containing the minimum and maximum values of latitude and longitude of the bounding box.
- **filter_out** (*boolean, optional*) – If set to false the function will return the trajectories points within the bounding box, and the points outside otherwise, by default False
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns dataframe with trajectories points filtered by bounding box or None

Return type DataFrame

pymove.preprocessing.filters.**by_datetime**(*move_data: DataFrame*, *start_datetime: str | None = None*, *end_datetime: str | None = None*, *filter_out: bool = False*, *inplace: bool = False*) → DataFrame | None

Filters trajectories points according to specified time range.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **start_datetime** (*str*) – The start date and time (Datetime format) of the time range, by default None
- **end_datetime** (*str*) – The end date and time (Datetime format) of the time range, by default None
- **filter_out** (*bool, optional*) – If set to true, the function will return the points of the trajectories with timestamp outside the time range. The points whithin the time range will be return if filter_out is False. by default False
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns dataframe with trajectories points filtered by time range or None

Return type DataFrame

pymove.preprocessing.filters.**by_id**(*move_data: DataFrame*, *id_: int | None = None*, *label_id: str = 'id'*, *filter_out: bool = False*, *inplace: bool = False*) → DataFrame | None

Filters trajectories points according to specified trajectory id.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **id** (*int*) – Specifies the number of the id used to filter the trajectories points
- **label_id** (*str, optional*) – The label of the column which contains the id of the trajectories, by default TRAJ_ID
- **filter_out** (*bool, optional*) – If set to true, the function will return the points of the trajectories with timestamp outside the time range. The points whithin the time range will be return if filter_out is False. by default False
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns dataframe with trajectories points filtered by id or None

Return type DataFrame

pymove.preprocessing.filters.**by_label**(*move_data: DataFrame*, *value: Any*, *label_name: str*, *filter_out: bool = False*, *inplace: bool = False*) → DataFrame | None

Filters trajectories points according to specified value and column label.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **value** (*The value to be use to filter the trajectories*) – Specifies the value used to filter the trajectories points
- **label_name** (*str*) – Specifies the label of the column used in the filtering
- **filter_out** (*bool, optional*) – If set to true, the function will return the points of the trajectories with timestamp outside the time range. The points whithin the time range will be return if filter_out is False. by default False
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns dataframe with trajectories points filtered by label or None

Return type DataFrame

pymove.preprocessing.filters.**by_tid**(*move_data: DataFrame*, *tid_: str | None = None*, *filter_out: bool = False*, *inplace: bool = False*) → DataFrame | None

Filters trajectories points according to a specified trajectory tid.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **tid** (*str*) – Specifies the number of the tid used to filter the trajectories points
- **label_tid** (*str, optional*) – The label of the column in the user dataframe which contains the tid of the trajectories, by default None
- **filter_out** (*bool, optional*) – If set to true, the function will return the points of the trajectories with timestamp outside the time range. The points whithin the time range will be return if filter_out is False. by default False
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns a dataframe with trajectories points filtered or None

Return type DataFrame

pymove.preprocessing.filters.**clean_consecutive_duplicates**(*move_data: DataFrame*, *subset: int | str | None = None*, *keep: str | bool = 'first'*, *inplace: bool = False*) → DataFrame | None

Removes consecutive duplicate rows of the Dataframe.

Optionally only certain columns can be consider.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **subset** (*Array of str, optional*) – Specifies Column label or sequence of labels, considered for identifying duplicates, by default None

- **keep** (*'first', 'last', optional*) – If keep is set as first, all the duplicates except for the first occurrence will be dropped. On the other hand if set to last, all duplicates except for the last occurrence will be dropped. If set to False, all duplicates are dropped. by default 'first'
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered, the duplicates will be dropped in place, otherwise a copy will be returned, by default False

Returns The filtered trajectories points without consecutive duplicates or None

Return type DataFrame

pymove.preprocessing.filters.**clean_gps_jumps_by_distance**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', jump_coefficient: float = 3.0, threshold: float = 1, label_dtype: Callable = <class 'numpy.float64'>, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Removes the trajectories points that are outliers from the dataframe.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **jump_coefficient** (*float, optional*) – by default 3
- **threshold** (*float, optional*) – Minimum value that the distance features must have in order to be considered outliers, by default 1
- **label_dtype** (*type, optional*) – Represents column id type, by default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories without the gps jumps or None

Return type DataFrame

pymove.preprocessing.filters.**clean_gps_nearby_points_by_distances**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_id: str = 'id'*, *radius_area: float = 10.0*, *label_dtype: Callable = <class 'numpy.float64'>*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Removes points from the trajectories with smaller distance from the point before.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **radius_area** (*float, optional*) – Species the minimum distance a point must have to it"srs previous point in order not to be dropped, by default 10
- **label_dtype** (*type, optional*) – Represents column id type, ,y default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, be default False

Returns The filtered trajectories without the gps nearby points by distance or None

Return type DataFrame

pymove.preprocessing.filters.**clean_gps_nearby_points_by_speed**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', speed_radius: float = 0.0, label_dtype: Callable = <class 'numpy.float64'>, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Removes points from the trajectories with smaller speed of travel.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, be defalt TRAJ_ID
- **speed_radius** (*float, optional*) – Species the minimum speed a point must have from it"srs previous point, in order not to be dropped, by default 0
- **label_dtype** (*type, optional*) – Represents column id type, by default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories without the gps nearby points by speed or None

Return type DataFrame

pymove.preprocessing.filters.**clean_gps_speed_max_radius**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', speed_max: float = 50.0, label_dtype: Callable = <class 'numpy.float64'>, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Removes trajectories points with higher speed.

Given any point p of the trajectory, the point will be removed if one of the following happens: if the travel speed from the point before p to p is greater than the max value of speed between adjacent points set by the user. Or the travel speed between point p and the next point is greater than the value set by the user. When the cleaning is done, the function will update the time and distance features in the dataframe and will call itself again. The function will finish processing when it can no longer find points disrespecting the limit of speed.

Parameters

- **move_data** (*dataframe*) – The input trajectory data

- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **speed_max** (*float, optional*) – Indicates the maximum value a point speed_to_prev and speed_to_next should have, in order not to be dropped, by default 50
- **label_dtype** (*type, optional*) – Represents column id type, by default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories without the gps nearby points or None

Return type DataFrame

pymove.preprocessing.filters.**clean_id_by_time_max**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', time_max: float = 3600, label_dtype: Callable = <class 'numpy.float64'>, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Clears GPS points with time by ID greater than a user-defined limit.

Parameters

- **move_data** (*dataframe.*) – The input data.
- **label_id** (*str, optional*) – The label of the column which contains the id of the trajectories, by default TRAJ_ID
- **time_max** (*float, optional*) – Indicates the maximum value time a set of points with the same id should have in order not to be dropped, by default 3600
- **label_dtype** (*type, optional*) – Represents column id type, by default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories with the maximum time.

Return type dataframe or None

pymove.preprocessing.filters.**clean_trajectories_short_and_few_points**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_id: str = 'tid'*, *min_trajectory_distance: float = 100*, *min_points_per_trajectory: int = 2*, *label_dtype: Callable = <class 'numpy.float64'>*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Eliminates from the given dataframe trajectories with fewer points and shorter length.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – The label of the column which contains the tid of the trajectories, by default TID
- **min_trajectory_distance** (*float, optional*) – Specifies the minimun length a trajectory must have in order not to be dropped, by default 100
- **min_points_per_trajectory** (*int, optional*) – Specifies the minimun number of points a trajectory must have in order not to be dropped, by default 2
- **label_dtype** (*type, optional*) – Represents column id type, by default np.float64.
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories with the minimum gps points and distance or None

Return type DataFrame

Notes

remove_tids_with_few_points must be performed before updating features.

pymove.preprocessing.filters.**clean_trajectories_with_few_points**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_tid: str = 'tid', min_points_per_trajectory: int = 2, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Removes from the given dataframe, trajectories with fewer points.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_tid** (*str, optional*) – The label of the column which contains the tid of the trajectories, by default TID
- **min_points_per_trajectory** (*int, optional*) – Specifies the minimum number of points a trajectory must have in order not to be dropped, by default 2
- **inplace** (*boolean, optional*) – if set to true the operation is done in place, the original dataframe will be altered and None is returned, by default False

Returns The filtered trajectories without the minimum number of gps points or None

Return type DataFrame

Raises KeyError – If the label feature is not in the dataframe

pymove.preprocessing.filters.**get_bbox_by_radius**(*coordinates: tuple[float, float], radius: float = 1000*) → tuple[float, float, float, float]

Defines minimum and maximum coordinates, given a distance radius from a point.

Parameters

- **coords** (*tuple (lat, lon)*) – The coordinates of point
- **radius** (*float, optional (1000 by default)*) –

Returns coordinates min and max of the bbox

Return type array

References

https://mathmesquita.me/2017/01/16/filtrando-localizacao-em-um-raio.html

pymove.preprocessing.segmentation module

Compression operations.

bbox_split, by_dist_time_speed, by_max_dist, by_max_time, by_max_speed

pymove.preprocessing.segmentation.**bbox_split**(*bbox: tuple[int, int, int, int], number_grids: int*) → DataFrame

Splits the bounding box in N grids of the same size.

Parameters

- **bbox** (*tuple*) – Tuple of 4 elements, containing the minimum and maximum values of latitude and longitude of the bounding box.
- **number_grids** (*int*) – Determines the number of grids to split the bounding box.

Returns Returns the latitude and longitude coordinates of the grids after the split.

Return type DataFrame

pymove.preprocessing.segmentation.**by_dist_time_speed**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', max_dist_between_adj_points: float = 3000, max_time_between_adj_points: float = 900, max_speed_between_adj_points: float = 50.0, drop_single_points: bool = True, label_new_tid: str = 'tid_part', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Splits the trajectories into segments based on distance, time and speed.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **max_dist_between_adj_points** (*float, optional*) – Specify the maximum distance a point should have from the previous point, in order not to be dropped, by default 3000
- **max_time_between_adj_points** (*float, optional*) – Specify the maximum travel time between two adjacent points, by default 900
- **max_speed_between_adj_points** (*float, optional*) – Specify the maximum speed of travel between two adjacent points, by default 50
- **drop_single_points** (*boolean, optional*) – If set to True, drops the trajectories with only one point, by default True
- **label_new_tid** (*str, optional*) – The label of the column containing the ids of the formed segments. Is the new splitted id, by default TID_PART

- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the aditional features: label_new_tid, that indicates the trajectory segment to which the point belongs to, by default False

Return type DataFrame

Note: Time, distance and speed features must be updated after split.

---

pymove.preprocessing.segmentation.**by_max_dist**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', max_dist_between_adj_points: float = 3000, drop_single_points: bool = True, label_new_tid: str = 'tid_dist', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Segments the trajectories based on distance.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **max_dist_between_adj_points** (*float, optional*) – Specify the maximum dist between two adjacent points, by default 3000
- **drop_single_points** (*boolean, optional*) – If set to True, drops the trajectories with only one point, by default True
- **label_new_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_DIST Is the new splitted id.
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the aditional features: label_segment, that indicates the trajectory segment to which the point belongs to.

Return type DataFrame

Note: Speed features must be updated after split.

---

pymove.preprocessing.segmentation.**by_max_speed**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', max_speed_between_adj_points: float = 50.0, drop_single_points: bool = True, label_new_tid: str = 'tid_speed', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Splits the trajectories into segments based on a maximum speed.

Parameters

- **move_data** (*dataframe.*) – The input trajectory data.

- **label_id** (*str, optional*) – Indicates the label of the id column in the users dataframe, by default TRAJ_ID
- **max_speed_between_adj_points** (*float, optional*) – Specify the maximum speed between two adjacent points, by default 50
- **drop_single_points** (*boolean, optional*) – If set to True, drops the trajectories with only one point, by default True
- **label_new_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_SPEED Is the new splitted id.
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the aditional features: label_segment, that indicates the trajectory segment to which the point belongs to

Return type DataFrame

Note: Speed features must be updated after split.

pymove.preprocessing.segmentation.**by_max_time**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_id: str = 'id', max_time_between_adj_points: float = 900.0, drop_single_points: bool = True, label_new_tid: str = 'tid_time', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Splits the trajectories into segments based on a maximum.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **label_id** (*str, optional*) – Indicates the label of the id column in the users dataframe, by default TRAJ_ID
- **max_time_between_adj_points** (*float, optional*) – Specify the maximum time between two adjacent points, by default 900
- **drop_single_points** (*boolean, optional*) – If set to True, drops the trajectories with only one point, by default True
- **label_new_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_TIME Is the new splitted id.
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the additional features: label_segment, that indicates the trajectory segment to which the point belongs to.

Return type DataFRame

Note: Speed features must be updated after split.

pymove.preprocessing.stay_point_detection module

Stop point detection operations.

create_or_update_move_stop_by_dist_time, create_or_update_move_and_stop_by_radius

pymove.preprocessing.stay_point_detection.**create_or_update_move_and_stop_by_radius**(*move_data*, *radius: float = 0*, *target_label: str = 'dist_to_prev'*, *new_label: str = 'situation'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Finds the stops and moves points of the dataframe.

If these points already exist, they will be updated.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **radius** (*float, optional*) – The radius value is used to determine if a segment is a stop. If the value of the point in target_label is greater than radius, the segment is a stop, otherwise it's a move, by default 0

- **target_label** (*String, optional*) – The feature used to calculate the stay points, by default DIST_TO_PREV
- **new_label** (*String, optional*) – Is the name of the column to indicates if a point is a stop of a move, by default SITUATION
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns dataframe with 2 aditional features: segment_stop and new_label. segment_stop indicates the trajectory segment to which the point belongs new_label indicates if the point represents a stop or moving point.

Return type DataFrame

pymove.preprocessing.stay_point_detection.**create_or_update_move_stop_by_dist_time**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', dist_radius: float = 30, time_radius: float = 900, label_id: str = 'id', new_label: str = 'segment_stop', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Determines the stops and moves points of the dataframe.

If these points already exist, they will be updated.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **dist_radius** (*float, optional*) – The first step in this function is segmenting the trajectory The segments are used to find the stop points The dist_radius defines the distance used in the segmentation, by default 30
- **time_radius** (*float, optional*) – The time_radius used to determine if a segment is a stop If the user stayed in the segment for a time greater than time_radius, than the segment is a stop, by default 900
- **label_id** (*str, optional*) – Indicates the label of the id column in the user dataframe, by default TRAJ_ID
- **new_label** (*float, optional*) – Is the name of the column to indicates if a point is a stop of a move, by default SEGMENT_STOP
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with 2 aditional features: segment_stop and stop. segment_stop indicates the trajectory segment to which the point belongs stop indicates if the point represents a stop.

Return type DataFrame

Module contents

Contains functions to preprocess the dataframes for manipulation.

compression, filters, segmentation, stay_point_detection

pymove.query package

Submodules

pymove.query.query module

Query operations.

range_query, knn_query

pymove.query.query.**knn_query**(*traj: pandas.core.frame.DataFrame*, *move_df: pandas.core.frame.DataFrame*, *k: int = 5*, *id_: str = 'id'*, *distance: str = 'MEDP'*, *latitude: str = 'lat'*, *longitude: str = 'lon'*, *datetime: str = 'datetime'*) → pandas.core.frame.DataFrame

Returns the k neighboring trajectories closest to the trajectory.

Given a k, a trajectory and a DataFrame with multiple paths.

Parameters

- **traj** (*dataframe*) – The input of one trajectory.
- **move_df** (*dataframe*) – The input trajectory data.
- **k** (*int, optional*) – neighboring trajectories, by default 5

- **id** (*str, optional*) – Label of the trajectories dataframe user id, by default TRAJ_ID
- **distance** (*string, optional*) – Distance measure type, by default MEDP
- **latitude** (*string, optional*) – Label of the trajectories dataframe referring to the latitude, by default LATITUDE
- **longitude** (*string, optional*) – Label of the trajectories dataframe referring to the longitude, by default LONGITUDE
- **datetime** (*string, optional*) – Label of the trajectories dataframe referring to the timestamp, by default DATETIME

Returns dataframe with near trajectories

Return type DataFrame

Raises ValueError: if distance measure is invalid

pymove.query.query.**range_query**(*traj: pandas.core.frame.DataFrame*, *move_df: pandas.core.frame.DataFrame*, *_id: str = 'id'*, *min_dist: float = 1000*, *distance: str = 'MEDP'*, *latitude: str = 'lat'*, *longitude: str = 'lon'*, *datetime: str = 'datetime'*) → pandas.core.frame.DataFrame

Returns all trajectories that have a distance equal to or less than the trajectory.

Given a distance, a trajectory, and a DataFrame with several trajectories.

Parameters

- **traj** (*dataframe*) – The input of one trajectory.
- **move_df** (*dataframe*) – The input trajectory data.
- **_id** (*str, optional*) – Label of the trajectories dataframe user id, by default TRAJ_ID
- **min_dist** (*float, optional*) – Minimum distance measure, by default 1000
- **distance** (*string, optional*) – Distance measure type, by default MEDP
- **latitude** (*string, optional*) – Label of the trajectories dataframe referring to the latitude, by default LATITUDE
- **longitude** (*string, optional*) – Label of the trajectories dataframe referring to the longitude, by default LONGITUDE
- **datetime** (*string, optional*) – Label of the trajectories dataframe referring to the timestamp, by default DATETIME

Returns dataframe with near trajectories

Return type DataFrame

Raises ValueError: if distance measure is invalid

Module contents

Contains functions to perform queries on trajectories.

query

pymove.semantic package

Submodules

pymove.semantic.semantic module

Semantic operations.

outliers create_or_update_out_of_the_bbox, create_or_update_gps_deactivated_signal, create_or_update_gps_jump, create_or_update_short_trajectory, create_or_update_gps_block_signal, filter_block_signal_by_repeated_amount_of_points, filter_block_signal_by_time, filter_longer_time_to_stop_segment_by_id

pymove.semantic.semantic.**create_or_update_gps_block_signal**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *max_time_stop: float = 7200*, *new_label: str = 'block_signal'*, *label_tid: str = 'tid_part'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Creates a new feature that inform segments with periods without moving.

Parameters

- **move_data** (*dataFrame*) – The input trajectories data.
- **max_time_stop** (*float, optional*) – Maximum time allowed with speed 0, by default 7200
- **new_label** (*string, optional*) – The name of the new feature with detected deactivated signals, by default BLOCK
- **label_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_PART Is the new slitted id.
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the additional features or None 'dist_to_prev', 'time_to_prev', 'speed_to_prev', 'tid_dist', 'block_signal'

Return type DataFrame

pymove.semantic.semantic.**create_or_update_gps_deactivated_signal**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *max_time_between_adj_points: float = 7200*, *new_label: str = 'deactivated_signal'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Creates a new feature that inform if point invalid.

If the max time between adjacent points is equal or less than max_time_between_adj_points.

Parameters

- **move_data** (*dataframe*) – The input trajectories data.
- **max_time_between_adj_points** (*float, optional*) – The max time between adjacent points, by default 7200
- **new_label** (*string, optional*) – The name of the new feature with detected deactivated signals, by default DEACTIVATED
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the additional features or None 'time_to_prev', 'time_to_next', 'time_prev_to_next', 'deactivate_signal'

Return type DataFrame

pymove.semantic.semantic.**create_or_update_gps_jump**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *max_dist_between_adj_points: float = 3000*, *new_label: str = 'gps_jump'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Creates a new feature that inform if point is a gps jump.

A jump is defined if the maximum distance between adjacent points is greater than max_dist_between_adj_points.

Parameters

- **move_data** (*dataframe*) – The input trajectories data.
- **max_dist_between_adj_points** (*float, optional*) – The maximum distance between adjacent points, by default 3000

- **new_label** (*string, optional*) – The name of the new feature with detected deactivated signals, by default GPS_JUMP
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the additional features or None 'dist_to_prev', 'dist_to_next', 'dist_prev_to_next', 'jump'

Return type DataFrame

pymove.semantic.semantic.**create_or_update_out_of_the_bbox**(*move_data: DataFrame, bbox: tuple[int, int, int, int], new_label: str = 'out_bbox', inplace: bool = False*) → DataFrame | None

Create or update a boolean feature to detect points out of the bbox.

Parameters

- **move_data** (*dataframe*) – The input trajectories data.
- **bbox** (*tuple*) – Tuple of 4 elements, containing the minimum and maximum values of latitude and longitude of the bounding box.
- **new_label** (*string, optional*) – The name of the new feature with detected points out of the bbox, by default OUT_BBOX
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns dataframe with a boolean feature with detected points out of the bbox, or None

Return type DataFrame

Raises ValueError – If feature generation fails

pymove.semantic.semantic.**create_or_update_short_trajectory**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', max_dist_between_adj_points: float = 3000, max_time_between_adj_points: float = 7200, max_speed_between_adj_points: float = 50, k_segment_max: int = 50, label_tid: str = 'tid_part', new_label: str = 'short_traj', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Creates a new feature that inform if point belongs to a short trajectory.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **max_dist_between_adj_points** (*float, optional*) – Specify the maximum distance a point should have from the previous point, in order not to be dropped, by default 3000
- **max_time_between_adj_points** (*float, optional*) – Specify the maximum travel time between two adjacent points, by default 7200
- **max_speed_between_adj_points** (*float, optional*) – Specify the maximum speed of travel between two adjacent points, by default 50
- **k_segment_max** (*int, optional*) – Specify the maximum number of segments in the trajectory, by default 50
- **label_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_PART
- **new_label** (*str, optional*) – The name of the new feature with short trajectories, by default SHORT
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns DataFrame with the aditional features or None 'dist_to_prev', 'time_to_prev', 'speed_to_prev', 'tid_part', 'short_traj'

Return type DataFrame

pymove.semantic.semantic.**filter_block_signal_by_repeated_amount_of_points**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *amount_max_of_points_stop: float = 30.0*, *max_time_stop: float = 7200*, *filter_out: bool = False*, *label_tid: str = 'tid_part'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Filters from dataframe points with blocked signal by amount of points.

Parameters

- **move_data** (*dataFrame*) – The input trajectories data.
- **amount_max_of_points_stop** (*float, optional*) – Maximum number of stopped points, by default 30
- **max_time_stop** (*float, optional*) – Maximum time allowed with speed 0, by default 7200
- **filter_out** (*boolean, optional*) – If set to True, it will return trajectory points with blocked signal, by default False
- **label_tid** (*str, optional*) – The label of the column containing the ids of the

formed segments, by default TID_PART

- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Filtered DataFrame with the additional features or None 'dist_to_prev', 'time_to_prev', 'speed_to_prev', 'tid_dist', 'block_signal'

Return type DataFrame

pymove.semantic.semantic.**filter_block_signal_by_time**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *max_time_stop: float = 7200*, *filter_out: bool = False*, *label_tid: str = 'tid_part'*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Filters from dataframe points with blocked signal by time.

Parameters

- **move_data** (*dataFrame*) – The input trajectories data.
- **max_time_stop** (*float, optional*) – Maximum time allowed with speed 0, by default 7200
- **filter_out** (*boolean, optional*) – If set to True, it will return trajectory points with blocked signal, by default False
- **label_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TID_PART
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Filtered DataFrame with the additional features or None 'dist_to_prev', 'time_to_prev', 'speed_to_prev', 'tid_dist', 'block_signal'

Return type DataFrame

pymove.semantic.semantic.**filter_longer_time_to_stop_segment_by_id**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *dist_radius: float = 30*, *time_radius: float = 900*, *label_id: str = 'id'*, *label_segment_stop: str = 'segment_stop'*, *filter_out: bool = False*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Filters from dataframe segment with longest stop time.

Parameters

- **move_data** (*dataFrame*) – The input trajectories data.
- **dist_radius** (*float, optional*) – The dist_radius defines the distance used in the segmentation, by default 30
- **time_radius** (*float, optional*) – The time_radius used to determine if a segment is a stop, by default 30 If the user stayed in the segment for a time greater than time_radius, than the segment is a stop.
- **label_tid** (*str, optional*) – The label of the column containing the ids of the formed segments, by default TRAJ_ID
- **label_segment_stop** (*str, optional*) – by default 'segment_stop'
- **filter_out** (*boolean, optional*) – If set to True, it will return trajectory points with longer time, by default True
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Filtered DataFrame with the additional features or None 'dist_to_prev', 'time_to_prev', 'speed_to_prev', 'tid_dist', 'block_signal'

Return type DataFrame

pymove.semantic.semantic.**outliers**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', jump_coefficient: float = 3.0, threshold: float = 1, new_label: str = 'outlier', inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Create or update a boolean feature to detect outliers.

Parameters

- **move_data** (*dataframe*) – The input trajectory data
- **jump_coefficient** (*float, optional*) – by default 3
- **threshold** (*float, optional*) – Minimum value that the distance features must have in order to be considered outliers, by default 1
- **new_label** (*string, optional*) – The name of the new feature with detected points out of the bbox, by default OUTLIER
- **inplace** (*bool, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns Returns a dataframe with the trajectories outliers or None

Return type DataFrame

Module contents

Contains semantic functions that adds new infomation to the trajectories.

semantic

pymove.uncertainty package

Submodules

pymove.uncertainty.privacy module

Not implemented.

pymove.uncertainty.reducing module

Not implemented.

Module contents

Contains functions to mask trajectories.

privacy, reducing

pymove.utils package

Submodules

pymove.utils.constants module

PyMove constants.

pymove.utils.conversions module

Unit conversion operations.

lat_meters, meters_to_eps, list_to_str, list_to_csv_str, list_to_svm_line, lon_to_x_spherical, lat_to_y_spherical, x_to_lon_spherical, y_to_lat_spherical, geometry_points_to_lat_and_lon, lat_and_lon_decimal_degrees_to_decimal, ms_to_kmh, kmh_to_ms, meters_to_kilometers, kilometers_to_meters, seconds_to_minutes, minute_to_seconds, minute_to_hours, hours_to_minute, seconds_to_hours, hours_to_seconds

pymove.utils.conversions.**geometry_points_to_lat_and_lon**(*move_data: pandas.core.frame.DataFrame*, *geometry_label: str = 'geometry'*, *drop_geometry: bool = False*, *inplace: bool = False*) → pandas.core.frame.DataFrame

Creates lat and lon columns from Points in geometry column.

Removes geometries that are not of the Point type.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data.
- **geometry** (*str, optional*) – Represents column name of the geometry column, by default GEOMETRY
- **drop_geometry** (*bool, optional*) – Option to drop the geometry column, by default False
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import geometry_points_to_lat_and_lon
>>> geom_points_df
    id                     geometry
0    1   POINT (116.36184 39.77529)
1    2   POINT (116.36298 39.77564)
2    3   POINT (116.33767 39.83148)
>>> geometry_points_to_lat_and_lon(geom_points_df)
    id                     geometry        lon       lat
0    1   POINT (116.36184 39.77529)  116.36184  39.77529
1    2   POINT (116.36298 39.77564)  116.36298  39.77564
2    3   POINT (116.33767 39.83148)  116.33767  39.83148
```

pymove.utils.conversions.**hours_to_minute**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_time: str = 'time_to_prev', new_label: str | None = None, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in hours, in label_distance column to minute.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import hours_to_minute
>>> geo_life_df
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153        ␣
↪0.000278        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788        ␣
↪0.001389         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083        ␣
↪0.001389         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671        ␣
↪0.001389         0.577934
>>> hours_to_minute(geo_life, inplace=False)
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153        ␣
↪0.016667        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788        ␣
↪0.083333         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083        ␣
↪0.083333         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671        ␣
↪0.083333         0.577934
```

pymove.utils.conversions.**hours_to_seconds**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_time: str = 'time_to_prev', new_label: str | None = None, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in hours, in label_distance column to seconds.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import hours_to_seconds
>>> geo_life_df
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN         ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153         ␣
↪0.000278        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788         ␣
↪0.001389         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083         ␣
↪0.001389         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671         ␣
↪0.001389         0.577934
>>> hours_to_seconds(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN         ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153         ␣
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788         ␣
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083         ␣
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671         ␣
↪    5.0         0.577934
```

pymove.utils.conversions.**kilometers_to_meters**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_distance: str = 'dist_to_prev', new_label: str | None = None, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in kilometers, in label_distance column to meters.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.

- **label_distance** (*str, optional*) – Represents column name of speed, by default DIST_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import kilometers_to_meters
>>> geo_life_df
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪     NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06             0.013690        ␣
↪     1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             0.007404        ␣
↪     5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             0.001821        ␣
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             0.002890        ␣
↪     5.0         0.577934
>>> kilometers_to_meters(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪     NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153        ␣
↪     1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788        ␣
↪     5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083        ␣
↪     5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671        ␣
↪     5.0         0.577934
```

pymove.utils.conversions.**kmh_to_ms**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame', label_speed: str = 'speed_to_prev', new_label: str | None = None, inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in kmh, in label_speed column to ms.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_speed** (*str, optional*) – Represents column name of speed, by default SPEED_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None

- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import kmh_to_ms
>>> geo_life_df
   id         lat          lon              datetime          dist_to_prev  ␣
↪time_to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                   NaN  ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06             13.690153  ␣
↪    1.0        49.284551
2   1   39.984224   116.319402   2008-10-23 05:53:11              7.403788  ␣
↪    5.0         5.330727
3   1   39.984211   116.319389   2008-10-23 05:53:16              1.821083  ␣
↪    5.0         1.311180
4   1   39.984217   116.319422   2008-10-23 05:53:21              2.889671  ␣
↪    5.0         2.080563
>>> kmh_to_ms(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                   NaN  ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06             13.690153  ␣
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11              7.403788  ␣
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16              1.821083  ␣
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21              2.889671  ␣
↪    5.0         0.577934
```

pymove.utils.conversions.**lat_and_lon_decimal_degrees_to_decimal**(*move_data: pandas.core.frame.DataFrame, latitude: str = 'lat', longitude: str = 'lon'*) → pandas.core.frame.DataFrame

Converts latitude and longitude format from decimal degrees to decimal format.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data.
- **latitude** (*str, optional*) – Represents column name of the latitude column, by default LATITUDE
- **longitude** (*str, optional*) – Represents column name of the longitude column, by default LONGITUDE

Returns A new dataframe with the converted feature

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import lat_and_lon_decimal_degrees_to_decimal
>>> lat_and_lon_df
   id     lat     lon
0   0   28.0N   94.8W
1   1   41.3N   50.4W
2   1   40.8N   47.5W
>>> lat_and_lon_decimal_degrees_to_decimal(lat_and_lon_df)
   id    lat      lon
0   0   28.0    -94.8
1   1   41.3    -50.4
2   1   40.8    -47.5
```

pymove.utils.conversions.**lat_meters**(*lat: float*) → float

Transform latitude degree to meters.

Parameters **lat** (*float*) – This represent latitude value.

Returns Represents the corresponding latitude value in meters.

Return type float

Examples

Latitude in Fortaleza: -3.71839 >>> from pymove.utils.conversions import lat_meters >>> lat_meters(-3.71839) 110832.75545918777

pymove.utils.conversions.**lat_to_y_spherical**(*lat: float | ndarray*) → float | ndarray

Convert latitude to Y EPSG:3857 WGS 84/Pseudo-Mercator.

Parameters **lat** (*float*) – This represents latitude value.

Returns Y offset from your original position in meters.

Return type float

Examples

```
>>> from pymove.utils.conversions import lat_to_y_spherical
>>> lat_fortaleza = -3.71839
>>> for_y = lat_to_y_spherical(lat_fortaleza)
>>> print(y_for, type(y_for))
-414220.15015607665 <class 'numpy.float64'>
```

References

https://epsg.io/transform

pymove.utils.conversions.**list_to_csv_str**(*input_list: list*) → str

Concatenates the elements of the list, joining them by ",".

Parameters **input_list** (*list*) – List with elements to be joined.

Returns Returns a string, resulting from concatenation of list elements, separeted by ",".

Return type str

Example

```
>>> from pymove.utils.conversions import list_to_csv_str
>>> list = [1,2,3,4,5]
>>> print(list_to_csv_str(list), type(list_to_csv_str(list)))
1,2,3,4,5 <class 'str'>
```

pymove.utils.conversions.**list_to_str**(*input_list: list*, *delimiter: str = ', '*) → str
Concatenates a list elements, joining them by the separator *delimiter*.

Parameters

- **input_list** (*list*) – List with elements to be joined.
- **delimiter** (*str, optional*) – The separator used between elements, by default ‘,’.

Returns Returns a string, resulting from concatenation of list elements, separeted by the delimiter.

Return type str

Example

```
>>> from pymove.utils.conversions import list_to_str
>>> list = [1,2,3,4,5]
>>> print(list_to_str(list, 'x'), type(list_to_str(list)))
1x2x3x4x5 <class 'str'>
```

pymove.utils.conversions.**list_to_svm_line**(*original_list: list*) → str
Concatenates list elements in consecutive element pairs.

Parameters **original_list** (*list*) – The elements to be joined

Returns Returns a string, resulting from concatenation of list elements in consecutive element pairs, separeted by ” “.

Return type str

Example

```
>>> from pymove.utils.conversions import list_to_svm_line
>>> list = [1,2,3,4,5]
>>> print(list_to_svm_line(list), type(list_to_svm_line(list)))
1 1:2 2:3 3:4 4:5 <class 'str'>
```

pymove.utils.conversions.**lon_to_x_spherical**(*lon: float | ndarray*) → float | ndarray
Convert longitude to X EPSG:3857 WGS 84/Pseudo-Mercator.

Parameters **lon** (*float*) – This represents longitude value.

Returns X offset from your original position in meters.

Return type float

Examples

```
>>> from pymove.utils.conversions import lon_to_x_spherical
>>> lon_fortaleza = -38.5434
>>> for_x = lon_to_x_spherical(lon_fortaleza)
>>> print(x_for, type(x_for))
-4290631.66144146 <class 'numpy.float64'>
```

References

https://epsg.io/transform

pymove.utils.conversions.**meters_to_eps**(*radius_meters: float*, *earth_radius: float = 6371*) → float

Converts radius in meters to eps.

Parameters

- **radius_meters** (*float*) – radius in meters
- **earth_radius** (*float, optional*) – radius of the earth in the location, by default EARTH_RADIUS

Returns radius in eps

Return type float

Example

```
>>> from pymove.utils.conversions import meters_to_eps
>>> earth_radius = 6371000
>>> meters_to_eps(earth_radius)
1000.0
```

pymove.utils.conversions.**meters_to_kilometers**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_distance: str = 'dist_to_prev'*, *new_label: str | None = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in meters, in label_distance column to kilometers.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_distance** (*str, optional*) – Represents column name of speed, by default DIST_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import meters_to_kilometers
>>> geo_life_df
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN       ␣
↪     NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153       ␣
↪     1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788       ␣
↪     5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083       ␣
↪     5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671       ␣
↪     5.0         0.577934
>>> meters_to_kilometers(geo_life, inplace=False)
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN       ␣
↪     NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06             0.013690       ␣
↪     1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             0.007404       ␣
↪     5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             0.001821       ␣
↪     5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             0.002890       ␣
↪     5.0         0.577934
```

pymove.utils.conversions.**minute_to_hours**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_time: str = 'time_to_prev'*, *new_label: str | None = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in minutes, in label_distance column to hours.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import minute_to_hours, seconds_to_minutes
>>> geo_life_df
```

(continues on next page)

(continued from previous page)

```
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪    5.0         0.577934
>>> seconds_to_minutes(geo_life, inplace=True)
>>> minute_to_hours(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪0.000278        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪0.001389         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪0.001389         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪0.001389         0.577934
```

pymove.utils.conversions.**minute_to_seconds**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_time: str = 'time_to_prev'*, *new_label: str | None = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in minutes, in label_distance column to seconds.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import minute_to_seconds
>>> geo_life_df
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
```

(continues on next page)

(continued from previous page)

```
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪0.016667        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪0.083333         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪0.083333         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪0.083333         0.577934
>>> minute_to_seconds(geo_life, inplace=False)
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪    5.0         0.577934
```

pymove.utils.conversions.**ms_to_kmh**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_speed: str = 'speed_to_prev'*, *new_label: str = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in ms, in label_speed column to kmh.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data.
- **label_speed** (*str, optional*) – Represents column name of speed, by default SPEED_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import ms_to_kmh
>>> geo_life_df
         lat          lon             datetime     id
0   39.984094   116.319236   2008-10-23 05:53:05     1
1   39.984198   116.319322   2008-10-23 05:53:06     1
2   39.984224   116.319402   2008-10-23 05:53:11     1
3   39.984211   116.319389   2008-10-23 05:53:16     1
4   39.984217   116.319422   2008-10-23 05:53:21     1
```

(continues on next page)

(continued from previous page)

```
>>> geo_life.generate_dist_time_speed_features(inplace=True)
>>> geo_life
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153        ␣
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788        ␣
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083        ␣
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671        ␣
↪    5.0         0.577934
>>> ms_to_kmh(geo_life, inplace=False)
   id         lat          lon             datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN        ␣
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153        ␣
↪    1.0        49.284551
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788        ␣
↪    5.0         5.330727
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083        ␣
↪    5.0         1.311180
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671        ␣
↪    5.0         2.080563
```

pymove.utils.conversions.**seconds_to_hours**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_time: str = 'time_to_prev'*, *new_label: str | None = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in seconds, in label_distance column to hours.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import minute_to_seconds, seconds_to_hours
>>> geo_life_df
```

(continues on next page)

(continued from previous page)

```
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪0.016667        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪0.083333         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪0.083333         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪0.083333         0.577934
>>> minute_to_seconds(geo_life, inplace=True)
>>> seconds_to_hours(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪0.000278        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪0.001389         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪0.001389         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪0.001389         0.577934
```

pymove.utils.conversions.**seconds_to_minutes**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *label_time: str = 'time_to_prev'*, *new_label: str | None = None*, *inplace: bool = False*) → 'PandasMoveDataFrame' | 'DaskMoveDataFrame' | None

Convert values, in seconds, in label_distance column to minutes.

Parameters

- **move_data** (*DataFame*) – Input trajectory data.
- **label_time** (*str, optional*) – Represents column name of speed, by default TIME_TO_PREV
- **new_label** (*str, optional*) – Represents a new column that will contain the conversion result, by default None
- **inplace** (*bool, optional*) – Whether the operation will be done in the original dataframe, by default False

Returns A new dataframe with the converted feature or None

Return type DataFrame

Example

```
>>> from pymove.utils.conversions import seconds_to_minutes
>>> geo_life_df
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
```

(continues on next page)

(continued from previous page)

```
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪    1.0        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪    5.0         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪    5.0         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪    5.0         0.577934
>>> seconds_to_minutes(geo_life, inplace=False)
   id         lat          lon              datetime         dist_to_prev    time_
↪to_prev    speed_to_prev
0   1   39.984094   116.319236   2008-10-23 05:53:05                  NaN
↪    NaN              NaN
1   1   39.984198   116.319322   2008-10-23 05:53:06            13.690153
↪0.016667        13.690153
2   1   39.984224   116.319402   2008-10-23 05:53:11             7.403788
↪0.083333         1.480758
3   1   39.984211   116.319389   2008-10-23 05:53:16             1.821083
↪0.083333         0.364217
4   1   39.984217   116.319422   2008-10-23 05:53:21             2.889671
↪0.083333         0.577934
```

pymove.utils.conversions.**x_to_lon_spherical**(*x: float | ndarray*) → float | ndarray

Convert X EPSG:3857 WGS 84 / Pseudo-Mercator to longitude.

Parameters **x** (*float*) – X offset from your original position in meters.

Returns Represents longitude.

Return type float

Examples

```
>>> from pymove.utils.conversions import x_to_lon_spherical
>>> for_x = -4290631.66144146
>>> print(x_to_lon_spherical(for_x), type(x_to_lon_spherical(for_x)))
-38.5434 <class 'numpy.float64'>
```

References

https://epsg.io/transform

pymove.utils.conversions.**y_to_lat_spherical**(*y: float | ndarray*) → float | ndarray

Convert Y EPSG:3857 WGS 84 / Pseudo-Mercator to latitude.

Parameters **y** (*float*) – Y offset from your original position in meters.

Returns Represents latitude.

Return type float

Examples

```
>>> from pymove.utils.conversions import y_to_lat_spherical
>>> for_y = -414220.15015607665
>>> print(y_to_lat_spherical(y_for), type(y_to_lat_spherical(y_for)))
-3.7183900000000096 <class 'numpy.float64'>
```

References

https://epsg.io/transform

pymove.utils.data_augmentation module

Data augmentation operations.

append_row, generate_trajectories_df, generate_start_feature, generate_destiny_feature, split_crossover, augmentation_trajectories_df, insert_points_in_df, instance_crossover_augmentation

pymove.utils.data_augmentation.**append_row**(*data: DataFrame*, *row: Series | None = None*, *columns: dict | None = None*)

Insert a new line in the dataframe with the information passed by parameter.

Parameters

- **data** (*DataFrame*) – The input trajectories data.
- **row** (*Series, optional*) – The row of a dataframe, by default None
- **columns** (*dict, optional*) – Dictionary containing the values to be added, by default None

pymove.utils.data_augmentation.**augmentation_trajectories_df**(*data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *restriction: str = 'destination only'*, *label_trajectory: str = 'trajectory'*, *insert_at_df: bool = False*, *frac: float = 0.5*) → DataFrame

Generates new data from unobserved trajectories, given a specific restriction.

By default, the algorithm uses the same route destination constraint.

Parameters

- **data** (*DataFrame*) – The input trajectories data.
- **restriction** (*str, optional*) – Constraint used to generate new data, by default 'destination only'
- **label_trajectory** (*str, optional*) – Label of the points sequences, by default TRAJECTORY
- **insert_at_df** (*boolean, optional*) – Whether to return a new DataFrame, by default False If True then value of copy is ignored.

- **frac** (*number, optional*) – Represents the percentage to be exchanged, by default 0.5

Returns Dataframe with the new data generated

Return type DataFrame

pymove.utils.data_augmentation.**generate_destiny_feature**(*data: pandas.core.frame.DataFrame*, *label_trajectory: str = 'trajectory'*)

Removes the first point from the trajectory and adds it in a new column 'start'.

Parameters

- **data** (*DataFrame*) – The input trajectory data.
- **label_trajectory** (*str, optional*) – Label of the points sequences, by default 'trajectory'

pymove.utils.data_augmentation.**generate_start_feature**(*data: pandas.core.frame.DataFrame*, *label_trajectory: str = 'trajectory'*)

Removes the last point from the trajectory and adds it in a new column 'destiny'.

Parameters

- **data** (*DataFrame*) – The input trajectory data.
- **label_trajectory** (*str, optional*) – Label of the points sequences, by default TRAJECTORY

pymove.utils.data_augmentation.**generate_trajectories_df**(*data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*) → DataFrame

Generates a dataframe with the sequence of location points of a trajectory.

Parameters **data** (*DataFrame*) – The input trajectory data.

Returns DataFrame of the trajectories

Return type DataFrame

pymove.utils.data_augmentation.**insert_points_in_df**(*data: pandas.core.frame.DataFrame*, *aug_df: pandas.core.frame.DataFrame*)

Inserts the points of the generated trajectories to the original data sets.

Parameters

- **data** (*DataFrame*) – The input trajectories data
- **aug_df** (*DataFrame*) – The data of unobserved trajectories

pymove.utils.data_augmentation.**instance_crossover_augmentation**(*data: pandas.core.frame.DataFrame*, *restriction: str = 'destination only'*, *label_trajectory: str = 'trajectory'*, *frac: float = 0.5*)

Generates new data from unobserved trajectories, with a specific restriction.

By default, the algorithm uses the same destination constraint as the route and inserts the points on the original dataframe.

Parameters

- **data** (*DataFrame*) – The input trajectories data
- **restriction** (*str, optional*) – Constraint used to generate new data, by default 'destination only'
- **label_trajectory** (*str, optional*) – Label of the points sequences, by default 'trajectory'
- **frac** (*number, optional*) – Represents the percentage to be exchanged, by default 0.5

pymove.utils.data_augmentation.**split_crossover**(*sequence_a: list*, *sequence_b: list*, *frac: float = 0.5*) → tuple[list, list]

Divides two arrays in the indicated ratio and exchange their halves.

Parameters

- **sequence_a** (*list or ndarray*) – Array any
- **sequence_b** (*list or ndarray*) – Array any
- **frac** (*float, optional*) – Represents the percentage to be exchanged, by default 0.5

Returns Arrays with the halves exchanged.

Return type Tuple[List, List]

pymove.utils.datetime module

Datetime operations.

date_to_str, str_to_datetime, datetime_to_str, datetime_to_min, min_to_datetime, to_day_of_week_int, working_day, now_str, deltatime_str, timestamp_to_millis, millis_to_timestamp, time_to_str, str_to_time, elapsed_time_dt, diff_time, create_time_slot_in_minute, generate_time_statistics, threshold_time_statistics

pymove.utils.datetime.**create_time_slot_in_minute**(*data: DataFrame*, *slot_interval: int = 15*, *initial_slot: int = 0*, *label_datetime: str = 'datetime'*, *label_time_slot: str = 'time_slot'*, *inplace: bool = False*) → DataFrame | None

Partitions the time in slot windows.

Parameters

- **data** (*DataFrame*) – dataframe with datetime column

- **slot_interval** (*int, optional*) – size of the slot window in minutes, by default 5
- **initial_slot** (*int, optional*) – initial window time, by default 0
- **label_datetime** (*str, optional*) – name of the datetime column, by default DATETIME
- **label_time_slot** (*str, optional*) – name of the time slot column, by default TIME_SLOT
- **inplace** (*boolean, optional*) – wether the operation will be done in the original dataframe, by default False

Returns data with converted time slots or None

Return type DataFrame

Examples

```
>>> from pymove.utils.datetime import create_time_slot_in_minute
>>> from pymove import datetime
>>> data
          lat          lon             datetime  id
0   39.984094   116.319236  2008-10-23 05:44:05   1
1   39.984198   116.319322  2008-10-23 05:56:06   1
2   39.984224   116.319402  2008-10-23 05:56:11   1
3   39.984224   116.319402  2008-10-23 06:10:15   1
>>> datetime.create_time_slot_in_minute(data, inplace=False)
          lat          lon             datetime  id   time_slot
0   39.984094   116.319236  2008-10-23 05:44:05   1          22
1   39.984198   116.319322  2008-10-23 05:56:06   1          23
2   39.984224   116.319402  2008-10-23 05:56:11   1          23
3   39.984224   116.319402  2008-10-23 06:10:15   1          24
```

pymove.utils.datetime.**date_to_str**(*dt: datetime.datetime*) → str

Get date, in string format, from timestamp.

Parameters **dt** (*datetime*) – This represents a date

Returns Represents the date in string format

Return type str

Example

```
>>> from datetime import datatime
>>> from pymove.utils.datetime import date_to_str
>>> time_now = datetime.now()
>>> print(time_now)
'2021-04-29 11:01:29.909340'
>>> print(type(time_now))
'<class 'datetime.datetime'>'
>>> print(date_to_str(time_now), type(time_now))
'2021-04-29   <class 'str'>'
```

pymove.utils.datetime.**datetime_to_min**(*dt: datetime.datetime*) → int

Converts a datetime to an int representation in minutes.

To do the reverse use: min_to_datetime.

Parameters **dt** (*datetime*) – Represents a datetime in datetime format

Returns Represents minutes from

Return type int

Example

```
>>> from pymove.utils.datetime import datetime_to_min
>>> from datetime import datetime
>>> time_now = datetime.now()
>>> print(type(datetime_to_min(time_now)))
'<class 'int'>'
>>> datetime_to_min(time_now)
'26996497'
```

pymove.utils.datetime.**datetime_to_str**(*dt: datetime.datetime*) → str

Converts a date in datetime format to string format.

Parameters **dt** (*datetime*) – Represents a datetime in datetime format.

Returns

- *str* – Represents a datetime in string format "%Y-%m-%d %H:%M:%S".
- *Example*
- ——-
- >>> *from pymove.utils.datetime import datetime_to_str*
- >>> *from datetime import datetime*
- >>> *time_now = datetime.now()*
- >>> *print(time_now)*
- **'2021-04-29 14** (*15:29.708113'*)
- >>> *print(type(time_now))*
- *'<class 'datetime.datetime'>'*
- >>> *print(datetime_to_str(time_now), type(datetime_to_str(time_now)))*
- **'2021-04-29 14** (*15:29 <class 'str' >'*)

pymove.utils.datetime.**deltatime_str**(*deltatime_seconds: float*) → str

Convert time in a format appropriate of time.

Parameters **deltatime_seconds** (*float*) – Represents the elapsed time in seconds

Returns **time_str** – Represents time in a format hh:mm:ss

Return type str

Examples

```
>>> from pymove.utils.datetime import deltatime_str
>>> deltatime_str(1082.7180936336517)
'18m:02.718s'
```

Notes

Output example if more than 24 hours: 25:33:57 https://stackoverflow.com/questions/3620943/measuring-elapsed-time-with-the-time-module

pymove.utils.datetime.**diff_time**(*start_time: datetime.datetime*, *end_time: datetime.datetime*) → int

Computes the elapsed time from the start time to the end time specified by the user.

Parameters

- **start_time** (*datetime*) – Specifies the start time of the time range to be computed
- **end_time** (*datetime*) – Specifies the start time of the time range to be computed

Returns Represents the time elapsed from the start time to the current time (when the function was called).

Return type int

Examples

```
>>> from datetime import datetime
>>> from pymove.utils.datetime import str_to_datetime
>>> time_now = datetime.now()
>>> start_time_1 = datetime(2020, 6, 29, 0, 0)
>>> start_time_2 = str_to_datetime('2020-06-29 12:45:59')
>>> print(diff_time(start_time_1, time_now))
26411808665
>>> print(diff_time(start_time_2, time_now))
26365849665
```

pymove.utils.datetime.**elapsed_time_dt**(*start_time: datetime.datetime*) → int

Computes the elapsed time from a specific start time.

Parameters **start_time** (*datetime*) – Specifies the start time of the time range to be computed

Returns Represents the time elapsed from the start time to the current time (when the function was called).

Return type int

Examples

```
>>> from datetime import datetime
>>> from pymove.utils.datetime import str_to_datetime
>>> start_time_1 = datetime(2020, 6, 29, 0, 0)
>>> start_time_2 = str_to_datetime('2020-06-29 12:45:59')
>>> print(elapsed_time_dt(start_time_1))
26411808666
>>> print(elapsed_time_dt(start_time_2))
26365849667
```

pymove.utils.datetime.**generate_time_statistics**(*data: pandas.core.frame.DataFrame*, *local_label: str = 'local_label'*)

Calculates time statistics of the pairwise local labels.

(average, standard deviation, minimum, maximum, sum and count) of the pairwise local labels of a symbolic trajectory.

Parameters

- **data** (*DataFrame*) – The input trajectories date.
- **local_label** (*str, optional*) – The name of the feature with local id, by default LOCAL_LABEL

Returns Statistics infomations of the pairwise local labels

Return type DataFrame

Example

```
>>> from pymove.utils.datetime import generate_time_statistics
>>> df
    local_label   prev_local   time_to_prev   id
0         house          NaN            NaN    1
1        market        house          720.0    1
2        market       market            5.0    1
3        market       market            1.0    1
4        school       market          844.0    1
>>> generate_time_statistics(df)
   local_label   prev_local    mean        std                 min          max  ␣
↪ sum      count
0        house       market   844.0   0.000000               844.0        844.0  ␣
↪844.0          1
1       market        house   720.0   0.000000               720.0        720.0  ␣
↪720.0          1
2       market       market     3.0   2.828427                 1.0          5.0  ␣
↪ 6.0          2
```

pymove.utils.datetime.**millis_to_timestamp**(*milliseconds: float*) → pandas._libs.tslibs.timestamps.Timestamp

Converts milliseconds to timestamp.

Parameters **milliseconds** (*int*) – Represents millisecond.

Returns Represents the date corresponding.

Return type Timestamp

Examples

```
>>> from pymove.utils.datetime import millis_to_timestamp
>>> millis_to_timestamp(1449907200123)
'2015-12-12 08:00:00.123000'
```

pymove.utils.datetime.**min_to_datetime**(*minutes: int*) → datetime.datetime

Converts an int representation in minutes to a datetime.

To do the reverse use: datetime_to_min.

Parameters **minutes** (*int*) – This represents a value in minutes

Returns Represents minutes in datetime format

Return type datetime

Example

```
>>> from pymove.utils.datetime import min_to_datetime
>>> print(min_to_datetime(26996497), type(min_to_datetime(26996497)))
'2021-04-30 13:37:00 <class 'datetime.datetime'>'
```

pymove.utils.datetime.**now_str**() → str
Get datetime of now.

Returns Represents a date

Return type str

Examples

```
>>> from pymove.utils.datetime import now_str
>>> now_str()
'2019-09-02 13:54:16'
```

pymove.utils.datetime.**str_to_datetime**(*dt_str: str*) → datetime.datetime
Converts a datetime in string format to datetime format.

Parameters **dt_str** (*str*) – Represents a datetime in string format, "%Y-%m-%d" or "%Y-%m-%d %H:%M:%S"

Returns Represents a datetime in datetime format

Return type datetime

Example

```
>>> from pymove.utils.datetime import str_to_datetime
>>> time_1 = '2020-06-29'
>>> time_2 = '2020-06-29 12:45:59'
>>> print(type(time_1), type(time_2))
'<class 'str'> <class 'str'>'
>>> print( str_to_datetime(time_1), type(str_to_datetime(time_1)))
'2020-06-29 00:00:00 <class 'datetime.datetime'>'
>>> print(str_to_datetime(time_2), type(str_to_datetime(time_2)))
'2020-06-29 12:45:59 <class 'datetime.datetime'>'
```

pymove.utils.datetime.**str_to_time**(*dt_str: str*) → datetime.datetime
Converts a time in string format "%H:%M:%S" to datetime format.

Parameters **dt_str** (*str*) – Represents a time in string format

Returns Represents a time in datetime format

Return type datetime

Examples

```
>>> from pymove.utils.datetime import str_to_time
>>> str_to_time("08:00:00")
datetime(1900, 1, 1, 8, 0)
```

pymove.utils.datetime.**threshold_time_statistics**(*df_statistics: DataFrame*, *mean_coef: float = 1.0*, *std_coef: float = 1.0*, *inplace: bool = False*) → DataFrame | None

Calculates and creates the threshold column.

The values are based in the time statistics dataframe for each segment.

Parameters

- **df_statistics** (*DataFrame*) – Time Statistics of the pairwise local labels.
- **mean_coef** (*float*) – Multiplication coefficient of the mean time for the segment, by default 1.0
- **std_coef** (*float*) – Multiplication coefficient of sdt time for the segment, by default 1.0
- **inplace** (*boolean, optional*) – wether the operation will be done in the original dataframe, by default False

Returns DataFrame of time statistics with the aditional feature: threshold, which indicates the time limit of the trajectory segment, or None

Return type DataFrame

Example

```
>>> from pymove.utils.datetime import generate_time_statistics
>>> df
    local_label   prev_local   time_to_prev   id
0         house          NaN            NaN    1
1        market        house          720.0    1
2        market       market            5.0    1
3        market       market            1.0    1
4        school       market          844.0    1
>>> statistics = generate_time_statistics(df)
>>> statistics
    local_label   prev_local    mean         std     min     max     sum   count
0         house       market   844.0    0.000000   844.0   844.0   844.0       1
1        market        house   720.0    0.000000   720.0   720.0   720.0       1
2        market       market     3.0    2.828427     1.0     5.0     6.0       2
>>> threshold_time_statistics(statistics)
    local_label   prev_local    mean         std     min                 max      ␣
↪    sum   count   threshold
0         house       market   844.0    0.000000   844.0               844.0      ␣
↪  844.0       1       844.0
1        market        house   720.0    0.000000   720.0               720.0      ␣
↪  720.0       1       720.0
2        market       market     3.0    2.828427     1.0                 5.0      ␣
↪    6.0       2         5.8
```

pymove.utils.datetime.**time_to_str**(*time: pandas._libs.tslibs.timestamps.Timestamp*) → str

Get time, in string format, from timestamp.

Parameters **time** (*Timestamp*) – Represents a time

Returns Represents the time in string format

Return type str

Examples

```
>>> from pymove.utils.datetime import time_to_str
>>> time_to_str("2015-12-12 08:00:00.123000")
'08:00:00'
```

pymove.utils.datetime.**timestamp_to_millis**(*timestamp: str*) → int

Converts a local datetime to a POSIX timestamp in milliseconds (like in Java).

Parameters **timestamp** (*str*) – Represents a date

Returns Represents millisecond results

Return type int

Examples

```
>>> from pymove.utils.datetime import timestamp_to_millis
>>> timestamp_to_millis('2015-12-12 08:00:00.123000')
1449907200123 (UTC)
```

pymove.utils.datetime.**to_day_of_week_int**(*dt: datetime.datetime*) → int

Get day of week of a date. Monday == 0... Sunday == 6.

Parameters **dt** (*datetime*) – Represents a datetime in datetime format.

Returns Represents day of week.

Return type int

Example

```
>>> from pymove.utils.datetime import str_to_datetime
>>> monday = str_to_datetime('2021-05-3 12:00:01')
>>> friday = str_to_datetime('2021-05-7 12:00:01')
>>> print(to_day_of_week_int(monday), type(to_day_of_week_int(monday)))
'0 <class 'int'>'
>>> print(to_day_of_week_int(friday), type(to_day_of_week_int(friday)))
'4 <class 'int'>'
```

pymove.utils.datetime.**working_day**(*dt: str | datetime*, *country: str = 'BR'*, *state: str | None = None*) → bool

Indices if a day specified by the user is a working day.

Parameters

- **dt** (*str or datetime*) – Specifies the day the user wants to know if it is a business day.
- **country** (*str*) – Indicates country to check for vacation days, by default 'BR'
- **state** (*str*) – Indicates state to check for vacation days, by default None

Returns if true, means that the day informed by the user is a working day. if false, means that the day is not a working day.

Return type boolean

Examples

```
>>> from pymove.utils.datetime import str_to_datetime
>>> independence_day = str_to_datetime('2021-09-7 12:00:01') # Holiday in Brazil
>>> next_day = str_to_datetime('2021-09-8 12:00:01') # Not a Holiday in Brazil
>>> print(working_day(independence_day, 'BR'))
False
>>> print(type(working_day(independence_day, 'BR')))
<class 'bool'>
>>> print(working_day(next_day, 'BR'))
True
>>> print(type(working_day(next_day, 'BR')))
'<class 'bool'>'
```

References

Countries and States names available in https://pypi.org/project/holidays/

pymove.utils.distances module

Distances operations.

haversine, euclidean_distance_in_meters, nearest_points, medp, medt

pymove.utils.distances.**euclidean_distance_in_meters**(*lat1: float | ndarray*, *lon1: float | ndarray*, *lat2: float | ndarray*, *lon2: float | ndarray*) → float | ndarray

Calculate the euclidean distance in meters between two points.

Parameters

- **lat1** (*float or array*) – latitute of point 1
- **lon1** (*float or array*) – longitude of point 1
- **lat2** (*float or array*) – latitute of point 2
- **lon2** (*float or array*) – longitude of point 2

Returns euclidean distance in meters between the two points.

Return type float or ndarray

Example

```
>>> from pymove.utils.distances import euclidean_distance_in_meters
>>> lat_fortaleza, lon_fortaleza = [-3.71839 ,-38.5434]
>>> lat_quixada, lon_quixada = [-4.979224744401671, -39.056434302570665]
>>> euclidean_distance_in_meters(
>>>     lat_fortaleza, lon_fortaleza, lat_quixada, lon_quixada
>>> )
151907.9670136588
```

pymove.utils.distances.**haversine**(*lat1: float | ndarray, lon1: float | ndarray, lat2: float | ndarray, lon2: float | ndarray, to_radians: bool = True, earth_radius: float = 6371*) → float | ndarray

Calculates the great circle distance between two points on the earth.

Specified in decimal degrees or in radians. All (lat, lon) coordinates must have numeric dtypes and be of equal length. Result in meters. Use 3956 in earth radius for miles.

Parameters

- **lat1** (*float or array*) – latitute of point 1
- **lon1** (*float or array*) – longitude of point 1
- **lat2** (*float or array*) – latitute of point 2
- **lon2** (*float or array*) – longitude of point 2
- **to_radians** (*boolean*) – Wether to convert the values to radians, by default True
- **earth_radius** (*int*) – Radius of sphere, by default EARTH_RADIUS

Returns Represents distance between points in meters

Return type float or ndarray

Example

```
>>> from pymove.utils.distances import haversine
>>> lat_fortaleza, lon_fortaleza = [-3.71839 ,-38.5434]
>>> lat_quixada, lon_quixada = [-4.979224744401671, -39.056434302570665]
>>> haversine(lat_fortaleza, lon_fortaleza, lat_quixada, lon_quixada)
151298.02548428564
```

References

Vectorized haversine function: https://stackoverflow.com/questions/43577086/pandas-calculate-haversine-distance-within-each-group-of-rows

About distance between two points: https://janakiev.com/blog/gps-points-distance-python/

pymove.utils.distances.**medp**(*traj1: pandas.core.frame.DataFrame, traj2: pandas.core.frame.DataFrame, latitude: str = 'lat', longitude: str = 'lon'*) → float

Returns the Mean Euclidian Distance Predictive between two trajectories.

Considers only the spatial dimension for the similarity measure.

Parameters

- **traj1** (*dataframe*) – The input of one trajectory.
- **traj2** (*dataframe*) – The input of another trajectory.
- **latitude** (*str, optional*) – Label of the trajectories dataframe referring to the latitude, by default LATITUDE
- **longitude** (*str, optional*) – Label of the trajectories dataframe referring to the longitude, by default LONGITUDE

Returns total distance

Return type float

Example

```
>>> from pymove.utils.distances import medp
>>> traj_1
            lat           lon            datetime     id
0   39.98471   116.319865   2008-10-23 05:53:23      1
>>> traj_2
            lat         lon              datetime     id
0   39.984674   116.31981   2008-10-23 05:53:28      1
>>> medp(traj_1, traj_2)
6.573431370981577e-05
```

pymove.utils.distances.**medt**(*traj1: pandas.core.frame.DataFrame*, *traj2: pandas.core.frame.DataFrame*, *latitude: str = 'lat'*, *longitude: str = 'lon'*, *datetime: str = 'datetime'*) → float

Returns the Mean Euclidian Distance Trajectory between two trajectories.

Considers the spatial dimension and the temporal dimension when measuring similarity.

Parameters

- **traj1** (*dataframe*) – The input of one trajectory.
- **traj2** (*dataframe*) – The input of another trajectory.
- **latitude** (*str, optional*) – Label of the trajectories dataframe referring to the latitude, by default LATITUDE
- **longitude** (*str, optional*) – Label of the trajectories dataframe referring to the longitude, by default LONGITUDE
- **datetime** (*str, optional*) – Label of the trajectories dataframe referring to the timestamp, by default DATETIME

Returns total distance

Return type float

Example

```
>>> from pymove.utils.distances import medt
>>> traj_1
         lat           lon              datetime  id
0   39.98471   116.319865   2008-10-23 05:53:23   1
>>> traj_2
          lat          lon              datetime  id
0   39.984674   116.31981   2008-10-23 05:53:28   1
>>> medt(traj_1, traj_2)
6.592419887747872e-05
```

pymove.utils.distances.**nearest_points**(*traj1: pandas.core.frame.DataFrame*, *traj2: pandas.core.frame.DataFrame*, *latitude: str = 'lat'*, *longitude: str = 'lon'*) → pandas.core.frame.DataFrame

Returns the point closest to another trajectory based on the Euclidean distance.

Parameters

- **traj1** (*dataframe*) – The input of one trajectory.
- **traj2** (*dataframe*) – The input of another trajectory.
- **latitude** (*str, optional*) – Label of the trajectories dataframe referring to the latitude, by default LATITUDE
- **longitude** (*str, optional*) – Label of the trajectories dataframe referring to the longitude, by default LONGITUDE

Returns dataframe with closest points

Return type DataFrame

Example

```
>>> from pymove.utils.distances import nearest_points
>>> df_a
         lat          lon             datetime    id
0   39.984198   116.319322   2008-10-23 05:53:06     1
1   39.984224   116.319402   2008-10-23 05:53:11     1
>>> df_b
          lat          lon             datetime    id
0   39.984211   116.319389   2008-10-23 05:53:16     1
1   39.984217   116.319422   2008-10-23 05:53:21     1
>>> nearest_points(df_a,df_b)
          lat          lon             datetime    id
0   39.984211   116.319389   2008-10-23 05:53:16     1
1   39.984211   116.319389   2008-10-23 05:53:16     1
```

pymove.utils.geoutils module

Geo operations.

v_color, create_geohash_df, create_bin_geohash_df, decode_geohash_to_latlon,

pymove.utils.geoutils.**create_bin_geohash_df**(*data: pandas.core.frame.DataFrame*, *precision: float = 15*)

Create trajectory geohash binaries and integrate with df.

Parameters

- **data** (*dataframe*) – The input trajectories data
- **precision** (*float, optional*) – Number of characters in resulting geohash, by default 15

Returns

Return type A DataFrame with the additional column 'bin_geohash'

Example

```
>>> from pymove.utils.geoutils import create_bin_geohash_df
>>> geoLife_df
         lat          lon
0   39.984094   116.319236
```

(continues on next page)

(continued from previous page)

```
1   39.984198   116.319322
2   39.984224   116.319402
3   39.984211   116.319389
4   39.984217   116.319422
>>> print(type(create_bin_geohash_df(geoLife_df)))
>>> geoLife_df
<class 'NoneType'>
          lat         lon                                            bin_geohash
0   39.984094   116.319236  [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, ...
1   39.984198   116.319322  [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, ...
2   39.984224   116.319402  [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, ...
3   39.984211   116.319389  [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, ...
4   39.984217   116.319422  [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, ...
```

pymove.utils.geoutils.**create_geohash_df**(*data: pandas.core.frame.DataFrame*, *precision: float = 15*)

Create geohash from geographic coordinates and integrate with df.

Parameters

- **data** (*dataframe*) – The input trajectories data
- **precision** (*float, optional*) – Number of characters in resulting geohash, by default 15

Returns

Return type A DataFrame with the additional column 'geohash'

Example

```
>>> from pymove.utils.geoutils import create_geohash_df, _reset_and_create_arrays_
↪none
>>> geoLife_df
          lat         lon
0   39.984094   116.319236
1   39.984198   116.319322
2   39.984224   116.319402
3   39.984211   116.319389
4   39.984217   116.319422
>>> print(type (create_geohash_df(geoLife_df)))
>>> geoLife_df
<class 'NoneType'>
          lat         lon           geohash
0   39.984094   116.319236   wx4eqyvh4xkg0xs
1   39.984198   116.319322   wx4eqyvhudszsev
2   39.984224   116.319402   wx4eqyvhyx8d9wc
3   39.984211   116.319389   wx4eqyvhyjnv5m7
4   39.984217   116.319422   wx4eqyvhyyr2yy8
```

pymove.utils.geoutils.**decode_geohash_to_latlon**(*data: pandas.core.frame.DataFrame*, *label_geohash: str = 'geohash'*, *reset_index: bool = True*)

Decode feature with hash of trajectories back to geographic coordinates.

Parameters

- **data** (*dataframe*) – The input trajectories data

- **label_geohash** (*str, optional*) – The name of the feature with hashed trajectories, by default GEOHASH
- **reset_index** (*boolean, optional*) – Condition to reset the df index, by default True

Returns

Return type A DataFrame with the additional columns 'lat_decode' and 'lon_decode'

Example

```
>>> from pymove.utils.geoutils import decode_geohash_to_latlon
>>> geoLife_df
          lat          lon           geohash
0   39.984094   116.319236   wx4eqyvh4xkg0xs
1   39.984198   116.319322   wx4eqyvhudszsev
2   39.984224   116.319402   wx4eqyvhyx8d9wc
3   39.984211   116.319389   wx4eqyvhyjnv5m7
4   39.984217   116.319422   wx4eqyvhyyr2yy8
>>> print(type(decode_geohash_to_latlon(geoLife_df)))
>>> geoLife_df
<class 'NoneType'>
          lat          lon           geohash  lat_decode   lon_decode
0   39.984094   116.319236   wx4eqyvh4xkg0xs   39.984094   116.319236
1   39.984198   116.319322   wx4eqyvhudszsev   39.984198   116.319322
2   39.984224   116.319402   wx4eqyvhyx8d9wc   39.984224   116.319402
3   39.984211   116.319389   wx4eqyvhyjnv5m7   39.984211   116.319389
4   39.984217   116.319422   wx4eqyvhyyr2yy8   39.984217   116.319422
```

pymove.utils.geoutils.**v_color**(*ob: shapely.geometry.base.BaseGeometry*) → str

Returns '#ffcc33' if object crosses otherwise it returns '#6699cc'.

Parameters **ob** (*geometry object*) – Any geometric object

Returns Geometric object color

Return type str

Example

```
>>> from pymove.utils.geoutils import v_color
>>> from shapely.geometry import LineString
>>> case_1 = LineString([(3,3),(4,4), (3,4)])
>>> case_2 = LineString([(3,3),(4,4), (4,3)])
>>> case_3 = LineString([(3,3),(4,4), (3,4), (4,3)])
>>> print(v_color(case_1), type(v_color(case_1)))
#6699cc <class 'str'>
>>> print(v_color(case_2), type(v_color(case_2)))
#6699cc <class 'str'>
>>> print(v_color(case_3), type(v_color(case_3)))
#ffcc33 <class 'str'>
```

pymove.utils.integration module

Integration operations.

union_poi_bank, union_poi_bus_station, union_poi_bar_restaurant, union_poi_parks, union_poi_police, join_collective_areas, join_with_pois, join_with_pois_by_category, join_with_events, join_with_event_by_dist_and_time, join_with_home_by_id, merge_home_with_poi

pymove.utils.integration.**join_collective_areas**(*data: DataFrame*, *areas: DataFrame*, *label_geometry: str = 'geometry'*, *inplace: bool = False*) → DataFrame | None

Performs the integration between trajectories and collective areas.

Generating a new column that informs if the point of the trajectory is inserted in a collective area.

Parameters

- **data** (*geopandas.GeoDataFrame*) – The input trajectory data
- **areas** (*geopandas.GeoDataFrame*) – The input coletive areas data
- **label_geometry** (*str, optional*) – Label referring to the Point of Interest category, by default GEOMETRY
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with joined geometries or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import join_collective_areas
>>> data
          lat           lon              datetime   id                    ⌐
↪geometry
0   39.984094   116.319236   2008-10-23 05:53:05    1    POINT (116.31924 39.
↪98409)
1   39.984198   116.319322   2008-10-23 05:53:06    1    POINT (116.31932 39.
↪98420)
2   39.984224   116.319402   2008-10-23 05:53:11    1    POINT (116.31940 39.
↪98422)
3   39.984211   116.319389   2008-10-23 05:53:16    1    POINT (116.31939 39.
↪98421)
4   39.984217   116.319422   2008-10-23 05:53:21    1    POINT (116.31942 39.
↪98422)
>>> area_c
         lat         lon               datetime  id                       ⌐
↪geometry
0  39.984094  116.319236    2008-10-23 05:53:05   1     POINT (116.319236 39.
↪984094)
1  40.006436  116.317701    2008-10-23 10:53:31   1     POINT (116.317701 40.
↪006436)
2  40.014125  116.306159    2008-10-23 23:43:56   1     POINT (116.306159 40.
↪014125)
3  39.984211  116.319389    2008-10-23 05:53:16   1     POINT (116.319389 39.
↪984211)
    POINT (116.32687 39.97901)
>>> join_collective_areas(gdf, area_c)
>>> gdf.head()
         lat         lon               datetime  id                         ⌐
↪geometry    violating
```

(continues on next page)

(continued from previous page)

```
0  39.984094  116.319236    2008-10-23 05:53:05   1         POINT (116.319236 39.
↪984094)          True
1  39.984198  116.319322    2008-10-23 05:53:06   1         POINT (116.319322 39.
↪984198)         False
2  39.984224  116.319402    2008-10-23 05:53:11   1         POINT (116.319402 39.
↪984224)         False
3  39.984211  116.319389    2008-10-23 05:53:16   1         POINT (116.319389 39.
↪984211)          True
4  39.984217  116.319422    2008-10-23 05:53:21   1         POINT (116.319422 39.
↪984217)         False
```

pymove.utils.integration.**join_with_event_by_dist_and_time**(*data: pandas.core.frame.DataFrame*, *df_events: pandas.core.frame.DataFrame*, *label_date: str = 'datetime'*, *label_event_id: str = 'event_id'*, *label_event_type: str = 'event_type'*, *time_window: float = 3600*, *radius: float = 1000*, *inplace: bool = False*)

Performs the integration between trajectories and events on windows.

Generating new columns referring to the category of the point of interest, the distance between the location of the user and location of the poi based on the distance and on time of each point of the trajectories.

Parameters

- **data** (*DataFrame*) – The input trajectory data.
- **df_pois** (*DataFrame*) – The input events points of interest data.
- **label_date** (*str, optional*) – Label of data referring to the datetime of the input trajectory data, by default DATETIME
- **label_event_id** (*str, optional*) – Label of df_events referring to the id of the event, by default EVENT_ID
- **label_event_type** (*str, optional*) – Label of df_events referring to the type of the event, by default EVENT_TYPE
- **time_window** (*float, optional*) – tolerable length of time range in 'seconds'for assigning the event's point of interest to the trajectory point, by default 3600
- **radius** (*float, optional*) – maximum radius of pois in *meters*, by default 1000
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import join_with_pois_by_dist_and_datetime
>>>  move_df
        lat         lon            datetime  id
```

(continues on next page)

(continued from previous page)

```
0  39.984094  116.319236 2008-10-23 05:53:05   1
1  39.984559  116.326696 2008-10-23 10:37:26   1
2  39.993527  116.326483 2008-10-24 00:02:14   2
3  39.978575  116.326975 2008-10-24 00:22:01   3
4  39.981668  116.310769 2008-10-24 01:57:57   3
>>> events
         lat         lon  id            datetime type_poi           name_poi
0  39.984094  116.319236   1 2008-10-23 05:53:05     show   forro_tropykalia
1  39.991013  116.326384   2 2008-10-23 10:27:26  corrida   racha_de_jumento
2  39.990013  116.316384   2 2008-10-23 10:37:26     show   dia_do_municipio
3  40.010000  116.312615   3 2008-10-24 01:57:57    feira  adocao_de_animais
>>> join_with_pois_by_dist_and_datetime(move_df, pois)
>>> move_df
         lat         lon            datetime  id                type_poi
↪dist_event                                  name_poi
0  39.984094  116.319236 2008-10-23 05:53:05   1                  [show]
↪     [0.0]                    [forro_tropykalia]
1  39.984559  116.326696 2008-10-23 10:37:26   1         [corrida, show]  [718.
↪144, 1067.53]  [racha_de_jumento, dia_do_municipio]
2  39.993527  116.326483 2008-10-24 00:02:14   2                    None
↪      None                                  None
3  39.978575  116.326975 2008-10-24 00:22:01   3                    None
↪      None                                  None
4  39.981668  116.310769 2008-10-24 01:57:57   3                    None
↪      None                                  None
```

Raises `ValueError` – If feature generation fails

`pymove.utils.integration.`**`join_with_events`**(*data: pandas.core.frame.DataFrame*, *df_events: pandas.core.frame.DataFrame*, *label_date: str = 'datetime'*, *time_window: int = 900*, *label_event_id: str = 'event_id'*, *label_event_type: str = 'event_type'*, *inplace: bool = False*)

Performs the integration between trajectories and the closest event in time window.

Generating new columns referring to the category of the point of interest, the distance from the nearest point of interest based on time of each point of the trajectories.

Parameters

- **`data`** (*`DataFrame`*) – The input trajectory data.
- **`df_events`** (*`DataFrame`*) – The input events points of interest data.
- **`label_date`** (*`str, optional`*) – Label of data referring to the datetime of the input trajectory data, by default DATETIME
- **`time_window`** (*`float, optional`*) – tolerable length of time range in *seconds* for assigning the event's point of interest to the trajectory point, by default 900
- **`label_event_id`** (*`str, optional`*) – Label of df_events referring to the id of the event, by default EVENT_ID
- **`label_event_type`** (*`str, optional`*) – Label of df_events referring to the type of the event, by default EVENT_TYPE
- **`inplace`** (*`boolean, optional`*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import join_with_events
>>>  move_df
         lat         lon            datetime  id
0  39.984094  116.319236 2008-10-23 05:53:05   1
1  39.984559  116.326696 2008-10-23 10:37:26   1
2  39.993527  116.326483 2008-10-24 00:02:14   2
3  39.978575  116.326975 2008-10-24 00:22:01   3
4  39.981668  116.310769 2008-10-24 01:57:57   3
>>> events
         lat         lon  id            datetime  event_type            event_id
0  39.984094  116.319236   1 2008-10-23 05:53:05        show     forro_tropykalia
1  39.991013  116.326384   2 2008-10-23 10:37:26        show     dia_do_municipio
2  40.010000  116.312615   3 2008-10-24 01:57:57       feira   adocao_de_animais
>>> join_with_events(move_df, events)
         lat         lon            datetime  id               event_type   dist_
↪event            event_id
0  39.984094  116.319236 2008-10-23 05:53:05   1                     show     0.
↪000000     forro_tropykalia
1  39.984559  116.326696 2008-10-23 10:37:26   1                     show   718.
↪144152     dia_do_municipio
2  39.993527  116.326483 2008-10-24 00:02:14   2                                 ⌐
↪  inf
3  39.978575  116.326975 2008-10-24 00:22:01   3                                 ⌐
↪  inf
4  39.981668  116.310769 2008-10-24 01:57:57   3                    feira  3154.
↪296880    adocao_de_animais
```

Raises `ValueError` – If feature generation fails

`pymove.utils.integration.`**`join_with_home_by_id`**(*data: pandas.core.frame.DataFrame, df_home: pandas.core.frame.DataFrame, label_id: str = 'id', label_address: str = 'formatted_address', label_city: str = 'city', drop_id_without_home: bool = False, inplace: bool = False*)

Performs the integration between trajectories and home points.

Generating new columns referring to the distance of the nearest home point, address and city of each trajectory point.

Parameters

- **`data`** (*`DataFrame`*) – The input trajectory data.
- **`df_home`** (*`DataFrame`*) – The input home points data.
- **`label_id`** (*`str, optional`*) – Label of df_home referring to the home point id, by default TRAJ_ID
- **`label_address`** (*`str, optional`*) – Label of df_home referring to the home point address, by default ADDRESS
- **`label_city`** (*`str, optional`*) – Label of df_home referring to the point city, by default CITY
- **`drop_id_without_home`** (*`bool, optional`*) – flag as an option to drop id's that don't have houses, by default False

- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import join_with_home_by_id
>>> move_df
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984559   116.326696  2008-10-23 10:37:26    1
2   40.002899   116.321520  2008-10-23 10:50:16    1
3   40.016238   116.307691  2008-10-23 11:03:06    1
4   40.013814   116.306525  2008-10-23 11:58:33    2
5   40.009735   116.315069  2008-10-23 23:50:45    2
>>> home_df
          lat          lon   id   formatted_address            city
0   39.984094   116.319236    1          rua da mae       quixiling
1   40.013821   116.306531    2      rua da familia   quixeramoling
>>> join_with_home_by_id(move_df, home_df)
>>> move_df
    id         lat          lon             datetime     dist_home
↪    home                city
0    1   39.984094   116.319236  2008-10-23 05:53:05      0.000000
↪rua da mae           quixiling
1    1   39.984559   116.326696  2008-10-23 10:37:26    637.690216
↪rua da mae           quixiling
2    1   40.002899   116.321520  2008-10-23 10:50:16   2100.053501
↪rua da mae           quixiling
3    1   40.016238   116.307691  2008-10-23 11:03:06   3707.066732
↪rua da mae           quixiling
4    2   40.013814   116.306525  2008-10-23 11:58:33      0.931101         rua
↪da familia       quixeramoling
5    2   40.009735   116.315069  2008-10-23 23:50:45    857.417540         rua
↪da familia       quixeramoling
```

pymove.utils.integration.**join_with_pois**(*data: pandas.core.frame.DataFrame, df_pois: pandas.core.frame.DataFrame, label_id: str = 'id', label_poi_name: str = 'name_poi', reset_index: bool = True, inplace: bool = False*)

Performs the integration between trajectories and the closest point of interest.

Generating two new columns referring to the name and the distance from the point of interest closest to each point of the trajectory.

Parameters

- **data** (*DataFrame*) – The input trajectory data.
- **df_pois** (*DataFrame*) – The input point of interest data.
- **label_id** (*str, optional*) – Label of df_pois referring to the Point of Interest id, by default TRAJ_ID
- **label_poi_name** (*str, optional*) – Label of df_pois referring to the Point of Interest name, by default NAME_POI
- **reset_index** (*bool, optional*) – Flag for reset index of the df_pois and data dataframes before the join, by default True

- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import join_with_pois
>>>  move_df
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984559   116.326696  2008-10-23 10:37:26    1
2   40.002899   116.321520  2008-10-23 10:50:16    1
3   40.016238   116.307691  2008-10-23 11:03:06    1
4   40.013814   116.306525  2008-10-23 11:58:33    2
5   40.009735   116.315069  2008-10-23 23:50:45    2
>>> pois
          lat          lon   id   type_poi             name_poi
0   39.984094   116.319236    1    policia        distrito_pol_1
1   39.991013   116.326384    2    policia       policia_federal
2   40.010000   116.312615    3   comercio   supermercado_aroldo
>>> join_with_pois(move_df, pois)
          lat          lon             datetime   id   id_poi            dist_
↪poi              name_poi
0   39.984094   116.319236  2008-10-23 05:53:05    1        1            0.
↪000000        distrito_pol_1
1   39.984559   116.326696  2008-10-23 10:37:26    1        1          637.
↪690216        distrito_pol_1
2   40.002899   116.321520  2008-10-23 10:50:16    1        3         1094.
↪860663   supermercado_aroldo
3   40.016238   116.307691  2008-10-23 11:03:06    1        3          810.
↪542998   supermercado_aroldo
4   40.013814   116.306525  2008-10-23 11:58:33    2        3          669.
↪973155   supermercado_aroldo
5   40.009735   116.315069  2008-10-23 23:50:45    2        3          211.
↪069129   supermercado_aroldo
```

pymove.utils.integration.**join_with_pois_by_category**(*data: pandas.core.frame.DataFrame, df_pois: pandas.core.frame.DataFrame, label_category: str = 'type_poi', label_id: str = 'id', inplace: bool = False*)

Performs the integration between trajectories and each type of points of interest.

Generating new columns referring to the category and distance from the nearest point of interest that has this category at each point of the trajectory.

Parameters

- **data** (*DataFrame*) – The input trajectory data.
- **df_pois** (*DataFrame*) – The input point of interest data.
- **label_category** (*str, optional*) – Label of df_pois referring to the point of interest category, by default TYPE_POI
- **label_id** (*str, optional*) – Label of df_pois referring to the point of interest id, by default TRAJ_ID

- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import join_with_pois_by_category
>>>  move_df
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984559   116.326696  2008-10-23 10:37:26    1
2   40.002899   116.321520  2008-10-23 10:50:16    1
3   40.016238   116.307691  2008-10-23 11:03:06    1
4   40.013814   116.306525  2008-10-23 11:58:33    2
5   40.009735   116.315069  2008-10-23 23:50:45    2
>>> pois
          lat          lon   id   type_poi             name_poi
0   39.984094   116.319236    1    policia        distrito_pol_1
1   39.991013   116.326384    2    policia       policia_federal
2   40.010000   116.312615    3   comercio   supermercado_aroldo
>>> join_with_pois_by_category(move_df, pois)
          lat          lon             datetime   id             id_policia
↪dist_policia   id_comercio   dist_comercio
0   39.984094   116.319236  2008-10-23 05:53:05    1                      1
↪  0.000000             3     2935.310277
1   39.984559   116.326696  2008-10-23 10:37:26    1                      1
↪637.690216             3     3072.696379
2   40.002899   116.321520  2008-10-23 10:50:16    1                      2
↪1385.087181            3     1094.860663
3   40.016238   116.307691  2008-10-23 11:03:06    1                      2
↪3225.288831            3      810.542998
4   40.013814   116.306525  2008-10-23 11:58:33    2                      2
↪3047.838222            3      669.973155
5   40.009735   116.315069  2008-10-23 23:50:45    2                      2
↪2294.075820            3      211.069129
```

pymove.utils.integration.**merge_home_with_poi**(*data: pandas.core.frame.DataFrame, label_dist_poi: str = 'dist_poi', label_name_poi: str = 'name_poi', label_id_poi: str = 'id_poi', label_home: str = 'home', label_dist_home: str = 'dist_home', drop_columns: bool = True, inplace: bool = False*)

Performs or merges the points of interest and the trajectories.

Considering the starting points as other points of interest, generating a new DataFrame.

Parameters

- **data** (*DataFrame*) – The input trajectory data, with join_with_pois and join_with_home_by_id applied.
- **label_dist_poi** (*str, optional*) – Label of data referring to the distance from the nearest point of interest, by default DIST_POI
- **label_name_poi** (*str, optional*) – Label of data referring to the name from the nearest point of interest, by default NAME_POI
- **label_id_poi** (*str, optional*) – Label of data referring to the id from the nearest point of interest, by default ID_POI

- **label_home** (*str, optional*) – Label of df_home referring to the home point, by default HOME
- **label_dist_home** (*str, optional*) – Label of df_home referring to the distance to the home point, by default DIST_HOME
- **drop_columns** (*bool, optional*) – Flag that controls the deletion of the columns referring to the id and the distance from the home point, by default
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Examples

```
>>> from pymove.utils.integration import (
>>>    merge_home_with_poi,
>>>    join_with_home_by_id
>>> )
>>> move_df
          lat          lon             datetime   id                    id_poi  ␣
↪    dist_poi          name_poi
0   39.984094   116.319236   2008-10-23 05:53:05    1                         1  ␣
↪    0.000000    distrito_pol_1
1   39.984559   116.326696   2008-10-23 10:37:26    1                         1  ␣
↪  637.690216    distrito_pol_1
2   40.002899   116.321520   2008-10-23 10:50:16    1                         2  ␣
↪ 1385.087181   policia_federal
3   40.016238   116.307691   2008-10-23 11:03:06    1                         2  ␣
↪ 3225.288831   policia_federal
4   40.013814   116.306525   2008-10-23 11:58:33    2                         2  ␣
↪ 3047.838222   policia_federal
5   40.009735   116.315069   2008-10-23 23:50:45    2                         2  ␣
↪ 2294.075820   policia_federal
>>> home_df
          lat          lon   id   formatted_address           city
0   39.984094   116.319236    1          rua da mae       quixiling
1   40.013821   116.306531    2      rua da familia   quixeramoling
>>> join_with_home_by_id(move, home_df, inplace=True)
>>> move_df
    id         lat          lon             datetime   id_poi      dist_poi      ␣
↪          name_poi    dist_home         home               city
0    1   39.984094   116.319236   2008-10-23 05:53:05        1      0.000000      ␣
↪    distrito_pol_1     0.000000        rua da mae       quixiling
1    1   39.984559   116.326696   2008-10-23 10:37:26        1    637.690216      ␣
↪    distrito_pol_1   637.690216        rua da mae       quixiling
2    1   40.002899   116.321520   2008-10-23 10:50:16        2   1385.087181      ␣
↪   policia_federal  2100.053501        rua da mae       quixiling
3    1   40.016238    16.307691   2008-10-23 11:03:06        2   3225.288831      ␣
↪   policia_federal  3707.066732        rua da mae       quixiling
4    2   40.013814   116.306525   2008-10-23 11:58:33        2   3047.838222      ␣
↪   policia_federal     0.931101    rua da familia   quixeramoling
5    2   40.009735   116.315069   2008-10-23 23:50:45        2   2294.075820      ␣
↪   policia_federal   857.417540    rua da familia   quixeramoling
>>> merge_home_with_poi(move_df)
    id         lat          lon             datetime           id_poi          ␣
↪dist_poi           name_poi            city
0    1   39.984094   116.319236   2008-10-23 05:53:05       rua da mae          ␣
↪0.000000               home       quixiling
```

(continues on next page)

(continued from previous page)

```
1    1   39.984559   116.326696   2008-10-23 10:37:26       rua da mae
↪637.690216               home       quixiling
2    1   40.002899   116.321520   2008-10-23 10:50:16                2
↪1385.087181    policia_federal       quixiling
3    1   40.016238   116.307691   2008-10-23 11:03:06                2
↪3225.288831    policia_federal       quixiling
4    2   40.013814   116.306525   2008-10-23 11:58:33   rua da familia
↪0.931101              home   quixeramoling
5    2   40.009735   116.315069   2008-10-23 23:50:45   rua da familia
↪857.417540               home   quixeramoling
```

pymove.utils.integration.**union_poi_bank**(*data: DataFrame*, *label_poi: str = 'type_poi'*, *banks: list[str] | None = None*, *inplace: bool = False*) → DataFrame | None

Performs the union between the different bank categories.

For Points of Interest in a single category named 'banks'.

Parameters

- **data** (*DataFrame*) – Input points of interest data
- **label_poi** (*str, optional*) – Label referring to the Point of Interest category, by default TYPE_POI
- **banks** (*list of str, optional*) –

  Names of poi refering to banks, by default banks = ['bancos_filiais', 'bancos_agencias', 'bancos_postos', 'bancos_PAE', 'bank',

  ]
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with poi or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import union_poi_bank
>>> pois_df
          lat          lon   id          type_poi
0   39.984094   116.319236    1              bank
1   39.984198   116.319322    2       randomvalue
2   39.984224   116.319402    3     bancos_postos
3   39.984211   116.319389    4       randomvalue
4   39.984217   116.319422    5        bancos_PAE
5   39.984710   116.319865    6     bancos_postos
6   39.984674   116.319810    7   bancos_agencias
7   39.984623   116.319773    8    bancos_filiais
8   39.984606   116.319732    9             banks
9   39.984555   116.319728   10             banks
>>> union_poi_bank(pois_df)
          lat          lon   id      type_poi
0   39.984094   116.319236    1         banks
1   39.984198   116.319322    2   randomvalue
```

(continues on next page)

(continued from previous page)

```
2   39.984224   116.319402    3         banks
3   39.984211   116.319389    4   randomvalue
4   39.984217   116.319422    5         banks
5   39.984710   116.319865    6         banks
6   39.984674   116.319810    7         banks
7   39.984623   116.319773    8         banks
8   39.984606   116.319732    9         banks
9   39.984555   116.319728   10         banks
```

pymove.utils.integration.**union_poi_bar_restaurant**(*data: DataFrame*, *label_poi: str = 'type_poi'*, *bar_restaurant: list[str] | None = None*, *inplace: bool = False*) → DataFrame | None

Performs the union between bar and restaurant categories.

For Points of Interest in a single category named 'bar-restaurant'.

Parameters

- **data** (*DataFrame*) – Input points of interest data
- **label_poi** (*str, optional*) – Label referring to the Point of Interest category, by default TYPE_POI
- **bar_restaurant** (*list of str, optional*) –

  Names of poi refering to bars or restaurants, by default

  bar_restaurant = ['restaurant', 'bar'

  ]
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with poi or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import union_poi_bar_restaurant
>>> pois_df
          lat          lon   id         type_poi
0   39.984094   116.319236    1       restaurant
1   39.984198   116.319322    2       restaurant
2   39.984224   116.319402    3      randomvalue
3   39.984211   116.319389    4              bar
4   39.984217   116.319422    5              bar
5   39.984710   116.319865    6   bar-restaurant
6   39.984674   116.319810    7        random123
7   39.984623   116.319773    8              123
>>> union_poi_bar_restaurant(pois_df)
          lat          lon   id         type_poi
0   39.984094   116.319236    1   bar-restaurant
1   39.984198   116.319322    2   bar-restaurant
2   39.984224   116.319402    3      randomvalue
3   39.984211   116.319389    4   bar-restaurant
4   39.984217   116.319422    5   bar-restaurant
```

(continues on next page)

(continued from previous page)

```
5   39.984710   116.319865    6   bar-restaurant
6   39.984674   116.319810    7        random123
7   39.984623   116.319773    8              123
```

pymove.utils.integration.**union_poi_bus_station**(*data: DataFrame*, *label_poi: str = 'type_poi'*, *bus_stations: list[str] | None = None*, *inplace: bool = False*) → DataFrame | None

Performs the union between the different bus station categories.

For Points of Interest in a single category named 'bus_station'.

Parameters

- **data** (*DataFrame*) – Input points of interest data
- **label_poi** (*str, optional*) – Label referring to the Point of Interest category, by default TYPE_POI
- **bus_stations** (*list of str, optional*) –

  Names of poi refering to bus_stations, by default

  bus_stations = ['transit_station', 'pontos_de_onibus'

  ]
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with poi or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import union_poi_bus_station
>>> pois_df
          lat          lon  id           type_poi
0   39.984094   116.319236   1    transit_station
1   39.984198   116.319322   2        randomvalue
2   39.984224   116.319402   3    transit_station
3   39.984211   116.319389   4   pontos_de_onibus
4   39.984217   116.319422   5    transit_station
5   39.984710   116.319865   6        randomvalue
6   39.984674   116.319810   7        bus_station
7   39.984623   116.319773   8        bus_station
>>> union_poi_bus_station(pois_df)
          lat          lon  id           type_poi
0   39.984094   116.319236   1        bus_station
1   39.984198   116.319322   2        randomvalue
2   39.984224   116.319402   3        bus_station
3   39.984211   116.319389   4        bus_station
4   39.984217   116.319422   5        bus_station
5   39.984710   116.319865   6        randomvalue
6   39.984674   116.319810   7        bus_station
7   39.984623   116.319773   8        bus_station
```

pymove.utils.integration.**union_poi_parks**(*data: DataFrame*, *label_poi: str = 'type_poi'*, *parks: list[str] | None = None*, *inplace: bool = False*) → DataFrame | None

Performs the union between park categories.

For Points of Interest in a single category named 'parks'.

Parameters

- **data** (*DataFrame*) – Input points of interest data
- **label_poi** (*str, optional*) – Label referring to the Point of Interest category, by default TYPE_POI
- **parks** (*list of str, optional*) –

  Names of poi refering to parks, by default

  parks = ['pracas_e_parques', 'park'

  ]
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with poi or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import union_poi_parks
>>> pois_df
          lat          lon   id          type_poi
0   39.984094   116.319236    1   pracas_e_parques
1   39.984198   116.319322    2              park
2   39.984224   116.319402    3             parks
3   39.984211   116.319389    4            random
4   39.984217   116.319422    5               123
5   39.984710   116.319865    6              park
6   39.984674   116.319810    7             parks
7   39.984623   116.319773    8   pracas_e_parques
>>> union_poi_parks(pois_df)
          lat          lon   id          type_poi
0   39.984094   116.319236    1             parks
1   39.984198   116.319322    2             parks
2   39.984224   116.319402    3             parks
3   39.984211   116.319389    4            random
4   39.984217   116.319422    5               123
5   39.984710   116.319865    6             parks
6   39.984674   116.319810    7             parks
7   39.984623   116.319773    8             parks
```

pymove.utils.integration.**union_poi_police**(*data: DataFrame*, *label_poi: str = 'type_poi'*, *police: list[str] | None = None*, *inplace: bool = False*) → DataFrame | None

Performs the union between police categories.

For Points of Interest in a single category named 'police'.

Parameters

- **data** (*DataFrame*) – Input points of interest data
- **label_poi** (*str, optional*) – Label referring to the Point of Interest category, by default TYPE_POI
- **police** (*list of str, optional*) –

  Names of poi refering to police stations, by default

  police = ['distritos_policiais', 'delegacia'

  ]
- **inplace** (*boolean, optional*) – if set to true the original dataframe will be altered to contain the result of the filtering, otherwise a copy will be returned, by default False

Returns data with poi or None

Return type DataFrame

Examples

```
>>> from pymove.utils.integration import union_poi_police
>>> pois_df
          lat          lon   id              type_poi
0   39.984094   116.319236    1   distritos_policiais
1   39.984198   116.319322    2                police
2   39.984224   116.319402    3                police
3   39.984211   116.319389    4   distritos_policiais
4   39.984217   116.319422    5                random
5   39.984710   116.319865    6           randomvalue
6   39.984674   116.319810    7                   123
7   39.984623   116.319773    8           bus_station
>>> union_poi_police(pois_df)
          lat          lon   id              type_poi
0   39.984094   116.319236    1                police
1   39.984198   116.319322    2                police
2   39.984224   116.319402    3                police
3   39.984211   116.319389    4                police
4   39.984217   116.319422    5                random
5   39.984710   116.319865    6           randomvalue
6   39.984674   116.319810    7                   123
7   39.984623   116.319773    8           bus_station
```

pymove.utils.log module

Logging operations.

progress_bar set_verbosity timer_decorator

pymove.utils.log.**progress_bar**(*sequence: Iterable*, *desc: str | None = None*, *total: int | None = None*, *miniters: int | None = None*)

Make and display a progress bar.

Parameters

- **sequence** (*iterable*) – Represents a sequence of elements.
- **desc** (*str, optional*) – Represents the description of the operation, by default None.

- **total** (*int, optional*) – Represents the total/number elements in sequence, by default None.
- **miniters** (*int, optional*) – Represents the steps in which the bar will be updated, by default None.

Returns

- *>>> from pymove.utils.log import progress_bar*
- *>>> for i in progress_bar(range(1,101), desc='Print 1 to 100')*
- *>>> print(i)*
- *# A bar that shows the progress of the iterations*

pymove.utils.log.**set_verbosity**(*level*)
Change logging level.

pymove.utils.log.**timer_decorator**(*func: Callable*) → Callable
A decorator that prints how long a function took to run.

pymove.utils.math module

Math operations.

is_number, std, avg_std, std_sample, avg_std_sample, arrays_avg, array_stats, interpolation

pymove.utils.math.**array_stats**(*values_array: list[float]*) → tuple[float, float, int]
Computes statistics about the array.

The sum of all the elements in the array, the sum of the square of each element and the number of elements of the array.

Parameters **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

Returns

- *float.* – The sum of all the elements in the array.
- *float* – The sum of the square value of each element in the array.
- *int.* – The number of elements in the array.

Example

```
>>> from pymove.utils.math import array_stats
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> print(array_stats(list), type(array_stats(list)))
(39.5, 327.25, 5) <class 'tuple'>
```

pymove.utils.math.**arrays_avg**(*values_array: list[float], weights_array: list[float] | None = None*) → float
Computes the mean of the elements of the array.

Parameters

- **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

- **weights_array** (*array, optional, default None.*) – Used to calculate the weighted average, indicates the weight of each element in the array (values_array).

Returns The mean of the array elements.

Return type float

Examples

```
>>> from pymove.utils.math import arrays_avg
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> weights = [0.1, 0.3, 0.15, 0.15, 0.3]
>>> print('standard average', arrays_avg(list), type(arrays_avg(list)))
'standard average 7.9 <class 'float'>'
>>> print(
>>>     'weighted average: ',
>>>      arrays_avg(list, weights),
>>>      type(arrays_avg(list, weights))
>>> )
'weighted average:  1.6979999999999997 <class 'float'>'
```

pymove.utils.math.**avg_std**(*values_array: list[float]*) → tuple[float, float]

Compute the average of standard deviation.

Parameters **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

Returns

- *float* – Represents the value of average.
- *float* – Represents the value of standard deviation.

Example

```
>>> from pymove.utils.math import avg_std
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> print(avg_std(list), type(avg_std(list)))
1.9493588689617927 <class 'float'>
```

pymove.utils.math.**avg_std_sample**(*values_array: list[float]*) → tuple[float, float]

Compute the average of standard deviation of sample.

Parameters **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

Returns

- *float* – Represents the value of average
- *float* – Represents the standard deviation of sample.

Example

```
>>> from pymove.utils.math import avg_std_sample
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> print(avg_std_sample(list), type(avg_std_sample(list)))
(7.9, 1.9493588689617927) <class 'tuple'>
```

pymove.utils.math.**interpolation**(*x0: float*, *y0: float*, *x1: float*, *y1: float*, *x: float*) → float

Performs interpolation.

Parameters

- **x0** (*float.*) – The coordinate of the first point on the x axis.
- **y0** (*float.*) – The coordinate of the first point on the y axis.
- **x1** (*float.*) – The coordinate of the second point on the x axis.
- **y1** (*float.*) – The coordinate of the second point on the y axis.
- **x** (*float.*) – A value in the interval (x0, x1).

Returns Is the interpolated or extrapolated value.

Return type float.

Example

```
>>> from pymove.utils.math import interpolation
>>> x0, y0, x1, y1, x = 2, 4, 3, 6, 3.5
>>> print(interpolation(x0,y0,x1,y1,x), type(interpolation(x0,y0,x1,y1,x)))
7.0 <class 'float'>
```

pymove.utils.math.**is_number**(*value: int | float | str*)

Returns if value is numerical or not.

Parameters **value** (*int, float, str*) –

Returns True if numerical, otherwise False

Return type boolean

Examples

```
>>> from pymove.utils.math import is_number
>>> a, b, c, d = 50, 22.5, '11.25', 'house'
>>> print(is_number(a), type(is_number(a)))
True <class 'bool'>
>>> print(is_number(b), type(is_number(b)))
True <class 'bool'>
>>> print(is_number(c), type(is_number(c)))
True <class 'bool'>
>>> print(is_number(d), type(is_number(d)))
False <class 'bool'>
```

pymove.utils.math.**std**(*values_array: list[float]*) → float

Compute standard deviation.

Parameters **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

Returns Represents the value of standard deviation.

Return type float

References

squaring with * is over 3 times as fast as with **2 http://stackoverflow.com/questions/29046346/comparison-of-power-to-multiplication-in-python

Example

```
>>> from pymove.utils.math import std
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> print(std(list), type(std(list)))
1.7435595774162693 <class 'float'>
```

pymove.utils.math.**std_sample**(*values_array: list[float]*) → float

Compute the standard deviation of sample.

Parameters **values_array** (*array like of numerical values.*) – Represents the set of values to compute the operation.

Returns Represents the value of standard deviation of sample.

Return type float

Example

```
>>> from pymove.utils.math import std_sample
>>> list = [7.8, 9.7, 6.4, 5.6, 10]
>>> print(std_sample(list), type(std_sample(list)))
1.9493588689617927 <class 'float'>
```

pymove.utils.mem module

Memory operations.

reduce_mem_usage_automatic, total_size, begin_operation, end_operation, sizeof_fmt, top_mem_vars

pymove.utils.mem.**begin_operation**(*name: str*) → dict

Gets the stats for the current operation.

Parameters **name** (*str*) – name of the operation

Returns dictionary with the operation stats

Return type dict

Examples

```
>>> from pymove.utils.mem import begin_operation
>>> operation = begin_operation('operation')
>>> operation
{
    'process': psutil.Process(
        pid=103401, name='python', status='running', started='21:48:11'
    ),
    'init': 293732352, 'start': 1622082973.8825781, 'name': 'operation'
}
```

pymove.utils.mem.**end_operation**(*operation: dict*) → dict

Gets the time and memory usage of the operation.

Parameters **operation** (*dict*) – dictionary with the begining stats of the operation

Returns dictionary with the operation execution stats

Return type dict

Examples

```
>>> import numpy as np
>>> import time
>>> from pymove.utils.mem import begin_operation, end_operation
>>> operation = begin_operation('create_arr')
>>> arr = np.arange(100000, dtype=np.float64)
>>> time.sleep(1.2)
>>> end_operation(operation)
{'name': 'create_arr', 'time in seconds': 1.2022554874420166, 'memory': '752.0 KiB
↪'}
```

pymove.utils.mem.**reduce_mem_usage_automatic**(*df: pandas.core.frame.DataFrame*)

Reduces the memory usage of the given dataframe.

df [dataframe] The input data to which the operation will be performed.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from pymove.utils.mem import reduce_mem_usage_automatic
>>> df = pd.DataFrame({'col_1': np.arange(10000, dtype=np.float64)})
>>> df.dtytes
col_1    float64
dtype: object
>>> reduce_mem_usage_automatic(df)
'Memory usage of dataframe is 0.08 MB'
'Memory usage after optimization is: 0.02 MB'
'Decreased by 74.9 %'
>>> df.dtytes
col_1    float16
dtype: object
```

pymove.utils.mem.**sizeof_fmt**(*mem_usage: float*, *suffix: str = 'B'*) → str

Returns the memory usage calculation of the last function.

Parameters

- **mem_usage** (*float*) – memory usage in bytes
- **suffix** (*string, optional*) – suffix of the unit, by default 'B'

Returns A string of the memory usage in a more readable format

Return type str

Examples

```
>>> from pymove.utils.mem import sizeof_fmt
>>> sizeof_fmt(1024)
1.0 KiB
>>> sizeof_fmt(2e6)
1.9 MiB
```

pymove.utils.mem.**top_mem_vars**(*variables: dict, n: int = 10, hide_private=True*) → pandas.core.frame.DataFrame

Shows the sizes of the active variables.

Parameters

- **variables** (*locals() or globals()*) – Whether to shows local or global variables
- **n** (*int, optional*) – number of variables to show, by default
- **hide_private** (*bool, optional*) – Whether to hide private variables, by default True

Returns dataframe with variables names and sizes

Return type DataFrame

Examples

```
>>> import numpy as np
>>> from pymove.utils.mem import top_mem_vars
>>> arr = np.arange(100000, dtype=np.float64)
>>> long_string = 'Hello World!' * 100
>>> top_mem_vars(locals())
            var        mem
0           arr  781.4 KiB
1   long_string    1.2 KiB
2         local    416.0 B
3  top_mem_vars    136.0 B
4            np     72.0 B
```

pymove.utils.mem.**total_size**(*o: object, handlers: Optional[dict] = None, verbose: bool = True*) → float

Calculates the approximate memory footprint of an given object.

Automatically finds the contents of the following builtin containers and their subclasses: tuple, list, deque, dict, set and frozenset.

Parameters

- **o** (*object*) – The object to calculate his memory footprint.
- **handlers** (*dict, optional*) –

  To search other containers, add handlers to iterate over their contents,

handlers = {SomeContainerClass: iter, OtherContainerClass: OtherContainerClass.get_elements}

by default None

- **verbose** (*boolean, optional*) – If set to True, the following information will be printed for each content of the object, by default False
  - the size of the object in bytes.
  - his type_
  - the object values

Returns The memory used by the given object

Return type float

Examples

```
>>> import numpy as np
>>> from pymove.utils.mem import total_size
>>> arr = np.arange(10000, dtype=np.float64)
>>> sz = total_size(arr)
'Size in bytes: 80104, Type: <class 'numpy.ndarray'>'
>>> sz
432
```

pymove.utils.trajectories module

Data operations.

read_csv, invert_dict, flatten_dict, flatten_columns, shift, fill_list_with_new_values, object_for_array, column_to_array

pymove.utils.trajectories.**column_to_array**(*data: pandas.core.frame.DataFrame*, *column: str*) → pandas.core.frame.DataFrame

Transforms all columns values to list.

Parameters

- **data** (*dataframe*) – The input trajectory data
- **column** (*str*) – Label of data referring to the column for conversion

Returns Dataframe with the selected column converted to list

Return type dataframe

Example

```
>>> from pymove.utils.trajectories import column_to_array
>>> move_df
          lat          lon             datetime  id   list_column
0   39.984094   116.319236  2008-10-23 05:53:05   1        '[1,2]'
1   39.984198   116.319322  2008-10-23 05:53:06   1        '[3,4]'
2   39.984224   116.319402  2008-10-23 05:53:11   1        '[5,6]'
3   39.984211   116.319389  2008-10-23 05:53:16   1        '[7,8]'
```

(continues on next page)

(continued from previous page)

```
4   39.984217   116.319422   2008-10-23 05:53:21   1       '[9,10]'
>>> column_to_array(move_df, column='list_column')
          lat          lon             datetime  id    list_column
0   39.984094   116.319236   2008-10-23 05:53:05   1      [1.0,2.0]
1   39.984198   116.319322   2008-10-23 05:53:06   1      [3.0,4.0]
2   39.984224   116.319402   2008-10-23 05:53:11   1      [5.0,6.0]
3   39.984211   116.319389   2008-10-23 05:53:16   1      [7.0,8.0]
4   39.984217   116.319422   2008-10-23 05:53:21   1     [9.0,10.0]
```

pymove.utils.trajectories.**fill_list_with_new_values**(*original_list: list*, *new_list_values: list*)

Copies elements from one list to another.

The elements will be positioned in the same position in the new list as they were in their original list.

Parameters

- **original_list** (*list.*) – The list to which the elements will be copied
- **new_list_values** (*list.*) – The list from which elements will be copied

Example

```
>>> from pymove.utils.trajectories import fill_list_with_new_values
>>> lst = [1, 2, 3, 4]
>>> fill_list_with_new_values(lt, ['a','b'])
>>> print(lst)
['a', 'b', 3, 4]
```

pymove.utils.trajectories.**flatten_columns**(*data: pandas.core.frame.DataFrame*, *columns: list*) → pandas.core.frame.DataFrame

Transforms columns containing dictionaries in individual columns.

Parameters

- **data** (*DataFrame*) – Dataframe with columns to be flattened
- **columns** (*list*) – List of columns from dataframe containing dictionaries

Returns Dataframe with the new columns from the flattened dictionary columns

Return type dataframe

References

https://stackoverflow.com/questions/51698540/import-nested-mongodb-to-pandas

Examples

```
>>> from pymove.utils.trajectories import flatten_columns
>>> move_df
          lat          lon             datetime  id           dict_column
0   39.984094   116.319236   2008-10-23 05:53:05   1             {'a': 1}
1   39.984198   116.319322   2008-10-23 05:53:06   1             {'b': 2}
2   39.984224   116.319402   2008-10-23 05:53:11   1     {'c': 3, 'a': 4}
3   39.984211   116.319389   2008-10-23 05:53:16   1             {'b': 2}
```

(continues on next page)

(continued from previous page)

```
4   39.984217   116.319422   2008-10-23 05:53:21   1      {'a': 3, 'c': 2}
>>> flatten_columns(move_df, columns='dict_column')
          lat             lon              datetime   id     dict_column_b
↪ dict_column_c   dict_column_a
0   39.984094      116.319236   2008-10-23 05:53:05    1               NaN
↪           NaN             1.0
1   39.984198      116.319322   2008-10-23 05:53:06    1               2.0
↪           NaN             NaN
2   39.984224      116.319402   2008-10-23 05:53:11    1               NaN
↪           3.0             4.0
3   39.984211      116.319389   2008-10-23 05:53:16    1               2.0
↪           NaN             NaN
4   39.984217      116.319422   2008-10-23 05:53:21    1               NaN
↪           2.0             3.0
```

pymove.utils.trajectories.**flatten_dict**(*d: dict*, *parent_key: str = ''*, *sep: str = '_'*) → dict

Flattens a nested dictionary.

Parameters

- **d** (*dict*) – Dictionary to be flattened
- **parent_key** (*str, optional*) – Key of the parent dictionary, by default ‘’
- **sep** (*str, optional*) – Separator for the parent and child keys, by default ‘_’

Returns Flattened dictionary

Return type dict

References

https://stackoverflow.com/questions/6027558/flatten-nested-dictionaries-compressing-keys

Examples

```
>>> from pymove.utils.trajectories import flatten_dict
>>> d = {'a': 1, 'b': {'c': 2, 'd': 3}}
>>> flatten_dict(d)
{'a': 1, 'b_c': 2, 'b_d': 3}
```

pymove.utils.trajectories.**invert_dict**(*d: dict*) → dict

Inverts the key:value relation of a dictionary.

Parameters **d** (*dict*) – dictionary to be inverted

Returns inverted dict

Return type dict

Examples

```
>>> from pymove.utils.trajectories import invert_dict
>>> traj_dict = {'a': 1, 'b': 2}
>>> invert_dict(traj_dict)
{1: 'a, 2: 'b'}
```

pymove.utils.trajectories.**object_for_array**(*object_: str*) → numpy.ndarray

Transforms an object into an array.

Parameters **object** (*str*) – object representing a list of integers or strings

Returns object converted to a list

Return type array

Examples

```
>>> from pymove.utils.trajectories import object_for_array
>>> list_str = '[1,2,3,4,5]'
>>> object_for_array(list_str)
array([1., 2., 3., 4., 5.], dtype=float32)
```

pymove.utils.trajectories.**read_csv**(*filepath_or_buffer: Union[PathLike[str], str, IO[AnyStr], io.RawIOBase, io.BufferedIOBase, io.TextIOBase, _io.TextIOWrapper, mmap.mmap], latitude: str = 'lat', longitude: str = 'lon', datetime: str = 'datetime', traj_id: str = 'id', type_: str = 'pandas', n_partitions: int = 1, **kwargs*)

Reads a *csv* file and structures the data.

Parameters

- **filepath_or_buffer** (*str or path object or file-like object*) – Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any os.PathLike. By file-like object, we refer to objects with a read() method, such as a file handle (e.g. via builtin open function) or StringIO.
- **latitude** (*str, optional*) – Represents the column name of feature latitude, by default 'lat'
- **longitude** (*str, optional*) – Represents the column name of feature longitude, by default 'lon'
- **datetime** (*str, optional*) – Represents the column name of feature datetime, by default 'datetime'
- **traj_id** (*str, optional*) – Represents the column name of feature id trajectory, by default 'id'
- **type** (*str, optional*) – Represents the type of the MoveDataFrame, by default 'pandas'
- **n_partitions** (*int, optional*) – Represents number of partitions for DaskMoveDataFrame, by default 1
- ****kwargs** (*Pandas read_csv arguments*) – https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html?highlight=read_csv#pandas.read_csv

Returns Trajectory data

Return type MoveDataFrameAbstract subclass

Examples

```
>>> from pymove.utils.trajectories import read_csv
>>> move_df = read_csv('geolife_sample.csv')
>>> move_df.head()
          lat          lon             datetime  id
0   39.984094   116.319236  2008-10-23 05:53:05   1
1   39.984198   116.319322  2008-10-23 05:53:06   1
2   39.984224   116.319402  2008-10-23 05:53:11   1
3   39.984211   116.319389  2008-10-23 05:53:16   1
4   39.984217   116.319422  2008-10-23 05:53:21   1
>>> type(move_df)
<class 'pymove.core.pandas.PandasMoveDataFrame'>
```

pymove.utils.trajectories.**shift**(*arr: list | Series | ndarray*, *num: int*, *fill_value: Any | None = None*) → ndarray

Shifts the elements of the given array by the number of periods specified.

Parameters

- **arr** (*array*) – The array to be shifted
- **num** (*int*) – Number of periods to shift. Can be positive or negative If positive, the elements will be pulled down, and pulled up otherwise
- **fill_value** (*float, optional*) – The scalar value used for newly introduced missing values, by default np.nan

Returns A new array with the same shape and **type_** as the initial given array, but with the indexes shifted.

Return type array

Notes

Similar to pandas shift, but faster.

References

https://stackoverflow.com/questions/30399534/shift-elements-in-a-numpy-array

Examples

```
>>> from pymove.utils.trajectories import shift
>>> array = [1, 2, 3, 4, 5, 6, 7]
>>> shift(array, 1)
[0 1 2 3 4 5 6]
>>> shift(array, 0)
[1, 2, 3, 4, 5, 6, 7]
>>> shift(array, -1)
[2 3 4 5 6 7 0]
```

pymove.utils.visual module

Visualization auxiliary operations.

add_map_legend, generate_color, rgb, hex_rgb, cmap_hex_color, get_cmap, save_wkt

pymove.utils.visual.**add_map_legend**(*m: Map, title: str, items: tuple | Sequence[tuple]*)

Adds a legend for a folium map.

Parameters

- **m** (*Map*) – Represents a folium map.
- **title** (*str*) – Represents the title of the legend
- **items** (*list of tuple*) – Represents the color and name of the legend items

References

https://github.com/python-visualization/folium/issues/528#issuecomment-421445303

Examples

```
>>> import folium
>>> from pymove.utils.visual import add_map_legend
>>> df
          lat          lon             datetime  id
0   39.984094   116.319236  2008-10-23 05:53:05   1
1   39.984198   116.319322  2008-10-23 05:53:06   1
2   39.984224   116.319402  2008-10-23 05:53:11   1
3   39.984211   116.319389  2008-10-23 05:53:16   2
4   39.984217   116.319422  2008-10-23 05:53:21   2
>>> m = folium.Map(location=[df.lat.median(), df.lon.median()])
>>> folium.PolyLine(mdf[['lat', 'lon']], color='red').add_to(m)
>>> pm.visual.add_map_legend(m, 'Color by ID', [(1, 'red')])
>>> m.get_root().to_dict()
{
    "name": "Figure",
    "id": "1d32230cd6c54b19b35ceaa864e61168",
    "children": {
        "map_6f1abc8eacee41e8aa9d163e6bbb295f": {
            "name": "Map",
            "id": "6f1abc8eacee41e8aa9d163e6bbb295f",
            "children": {
                "openstreetmap": {
                    "name": "TileLayer",
                    "id": "f58c3659fea348cb828775f223e1e6a4",
                    "children": {}
                },
                "poly_line_75023fd7df01475ea5e5606ddd7f4dd2": {
                    "name": "PolyLine",
                    "id": "75023fd7df01475ea5e5606ddd7f4dd2",
                    "children": {}
                }
            }
        },
        "map_legend": {  # legend element
```

(continues on next page)

(continued from previous page)

```
            "name": "MacroElement",
            "id": "72911b4418a94358ba8790aab93573d1",
            "children": {}
        }
    },
    "header": {
        "name": "Element",
        "id": "e46930fc4152431090b112424b5beb6a",
        "children": {
            "meta_http": {
                "name": "Element",
                "id": "868e20baf5744e82baf8f13a06849ecc",
                "children": {}
            }
        }
    },
    "html": {
        "name": "Element",
        "id": "9c4da9e0aac349f594e2d23298bac171",
        "children": {}
    },
    "script": {
        "name": "Element",
        "id": "d092078607c04076bf58bd4593fa1684",
        "children": {}
    }
}
```

pymove.utils.visual.**cmap_hex_color**(*cmap: matplotlib.colors.ListedColormap*, *i: int*) → str

Convert a Colormap to hex color.

Parameters

- **cmap** (*ListedColormap*) – Represents the Colormap
- **i** (*int*) – List color index

Returns Represents corresponding hex str

Return type str

Examples

```
>>> from pymove.utils.visual import  cmap_hex_color
>>> import matplotlib.pyplot as plt
>>> jet = plt.get_cmap('jet')  # This comand generates a Linear Segmented Colormap
>>> print(cmap_hex_color(jet, 0))
'#000080'
>>> print(cmap_hex_color(jet, 1))
'#000084'
```

pymove.utils.visual.**generate_color**() → str

Generates a random color.

Returns

Return type Random HEX color

Examples

```
>>> from pymove.utils.visual import generate_color
>>> print(generate_color(), type(generate_color()))
'#E0FFFF' <class 'str'>
>>> print(generate_color(), type(generate_color()))
'#808000' <class 'str'>
```

pymove.utils.visual.**get_cmap**(*cmap: str*) → matplotlib.colors.Colormap
Returns a matplotlib colormap instance.

Parameters **cmap** (*str*) – name of the colormar

Returns matplotlib colormap

Return type Colormap

Examples

```
>>> from pymove.utils.visual import  get_cmap
>>> print(get_cmap('Greys')
<matplotlib.colors.LinearSegmentedColormap object at 0x7f743fc04bb0>
```

pymove.utils.visual.**hex_rgb**(*rgb_colors: tuple[float, float, float]*) → str
Return a hex str, as used in Tk plots.

Parameters **rgb_colors** (*tuple*) – Represents a tuple with three positions that correspond to the percentage red, green and blue colors

Returns Represents a color in hexadecimal format

Return type str

Examples

```
>>> from pymove.utils.visual import hex_rgb
>>> print(hex_rgb((0.1, 0.2, 0.7)), type(hex_rgb((0.1, 0.2, 0.7))))
'#33B219' <class 'str'>
>>> print(hex_rgb((0.5, 0.4, 0.1)), type(hex_rgb((0.5, 0.4, 0.1))))
'#66197F' <class 'str'>
```

pymove.utils.visual.**randint**(*low*, *high=None*, *size=None*, *dtype=int*)
Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Note: New code should use the `integers` method of a `default_rng()` instance instead; please see the random-quick-start.

Parameters

- **low** (*int or array-like of ints*) – Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

- **high** (*int or array-like of ints, optional*) – If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **dtype** (*dtype, optional*) – Desired dtype of the result. Byteorder must be native. The default value is int.

  New in version 1.11.0.

Returns **out** – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type int or ndarray of ints

See also:

random_integers() similar to *randint*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

Generator.integers() which should be used for new code.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

pymove.utils.visual.**rgb**(*rgb_colors: tuple[float, float, float]*) → tuple[int, int, int]

Return a tuple of integers, as used in AWT/Java plots.

Parameters **rgb_colors** (*tuple*) – Represents a tuple with three positions that correspond to the percentage red, green and blue colors.

Returns Represents a tuple of integers that correspond the colors values.

Return type tuple

Examples

```
>>> from pymove.utils.visual import rgb
>>> print(rgb((0.1, 0.2, 0.7)), type(rgb((0.1, 0.2, 0.7))))
(51, 178, 25) <class 'tuple'>
>>> print(rgb((0.5, 0.4, 0.1)), type(rgb((0.5, 0.4, 0.1))))
(102, 25, 127) <class 'tuple'>
```

pymove.utils.visual.**save_wkt**(*move_data: pandas.core.frame.DataFrame*, *filename: str*, *label_id: str = 'id'*)

Save a visualization in a map in a new file .wkt.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **filename** (*str*) – Represents the filename
- **label_id** (*str*) – Represents column name of trajectory id

Returns **File**

Return type A file.wkt that contains geometric points that build a map visualization

Examples

```
>>> from pymove.utils.visual import save_wkt
>>> df.head()
          lat          lon             datetime  id
0   39.984094   116.319236  2008-10-23 05:53:05   1
1   39.984198   116.319322  2008-10-23 05:53:06   1
2   39.984224   116.319402  2008-10-23 05:53:11   1
3   39.984211   116.319389  2008-10-23 05:53:16   2
4   39.984217   116.319422  2008-10-23 05:53:21   2
>>> save_wkt(df, 'test.wkt', 'id')
>>> with open('test.wtk') as f:
>>>     print(f.read())
'id;linestring'
'1;LINESTRING(116.319236 39.984094,116.319322 39.984198,116.319402 39.984224)'
'2;LINESTRING(116.319389 39.984211,116.319422 39.984217)'
```

Module contents

Contains utility functions.

constants, conversions, data_augmentation, datetime, distances, geoutils, integration, log, math, mem, trajectories, visual

pymove.visualization package

Submodules

pymove.visualization.folium module

Folium operations.

save_map, create_base_map, heatmap, heatmap_with_time, cluster, faster_cluster, plot_markers, plot_trajectories, plot_trajectory_by_id, plot_trajectory_by_period, plot_trajectory_by_day_week, plot_trajectory_by_date, plot_trajectory_by_hour, plot_stops, plot_bbox, plot_points, plot_poi, plot_event, plot_traj_timestamp_geo_json

pymove.visualization.folium.**cluster**(*move_data: DataFrame*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *filename: str = 'cluster.html'*) → Map

Generate visualization of Heat Map using folium plugin.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **zoom_start** (*float, optional*) – Initial zoom level for the map, by default 12
- **radius** (*float, optional*) – Radius of each "point" of the heatmap, by default 8
- **base_map** (*Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin and zoom_start, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'cluster.html'

Returns folium Map

Return type Map

Examples

```
>>> from pymove.visualization.folium import cluster
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
```

(continues on next page)

(continued from previous page)

```
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> cluster(move_df)
```

pymove.visualization.folium.**create_base_map**(*move_data: DataFrame, lat_origin: float | None = None, lon_origin: float | None = None, tile: str = 'CartoDB positron', default_zoom_start: float = 12*) → Map

Generates a folium map.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **default_zoom_start** (*float, optional*) – Represents the zoom which will be the center of the map, by default 12

Returns a folium map

Return type Map

Examples

```
>>> from pymove.visualization.folium import create_base_map
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> create_base_map(move_df)
```

pymove.visualization.folium.**faster_cluster**(*move_data: DataFrame, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, filename: str = 'faster_cluster.html'*) → Map

Generate visualization of Heat Map using folium plugin.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None

- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **zoom_start** (*float, optional*) – Initial zoom level for the map, by default 12
- **radius** (*float, optional*) – Radius of each "point" of the heatmap, by default 8
- **base_map** (*Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin and zoom_start, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'faster_cluster.html'

Returns folium Map

Return type Map

Examples

```
>>> from pymove.visualization.folium import faster_cluster
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> faster_cluster(move_df)
```

pymove.visualization.folium.**heatmap**(*move_data: DataFrame, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, radius: float = 8, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, filename: str = 'heatmap.html'*) → Map

Generate visualization of Heat Map using folium plugin.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **zoom_start** (*float, optional*) – Initial zoom level for the map, by default 12
- **radius** (*float, optional*) – Radius of each "point" of the heatmap, by default 8

- **base_map** (*Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin and zoom_start, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'heatmap.html'

Returns folium Map

Return type Map

Examples

```
>>> from pymove.visualization.folium import heatmap
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> heatmap(move_df)
```

pymove.visualization.folium.**heatmap_with_time**(*move_data: DataFrame, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, radius: float = 8, min_opacity: float = 0.5, max_opacity: float = 0.8, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, filename: str = 'heatmap_time.html'*) → Map

Generate visualization of Heat Map using folium plugin.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **zoom_start** (*float, optional*) – Initial zoom level for the map, by default 12
- **radius** (*float, optional*) – Radius of each "point" of the heatmap, by default 8
- **min_opacity** (*float, optional*) – Minimum heat opacity, by default 0.5.
- **max_opacity** (*float, optional*) – Maximum heat opacity, by default 0.8.

- **base_map** (*Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin and zoom_start, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'heatmap_time.html'

Returns folium Map

Return type Map

Examples

```
>>> from pymove.visualization.folium import heatmap_with_time
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> heatmap_with_time(move_df)
```

pymove.visualization.folium.**plot_bbox**(*bbox_tuple: tuple[float, float, float, float], base_map: Map | None = None, tiles: str = 'CartoDB positron', color: str = 'red', save_as_html: bool = False, filename: str = 'bbox.html'*) → Map

Plots a bbox using Folium.

Parameters

- **bbox_tuple** (*tuple.*) – Represents a bound box, that is a tuple of 4 values with the min and max limits of latitude e longitude.
- **base_map** (*Folium map, optional*) – A folium map to plot the trajectories. If None a map will be created, by default None.
- **tiles** (*str, optional*) – by default TILES[0]
- **color** (*str, optional*) – Represents color of lines on map, by default 'red'.
- **file** (*str, optional*) – Represents filename, by default 'bbox.html'.
- **save_map** (*Boolean, optional*) – Wether to save the bbox folium map, by default False.

Returns folium map with bounding box

Return type Map

Examples

```
>>> from pymove.visualization.folium import plot_bbox
>>> plot_bbox((39.984094,116.319236,39.997535,116.196345))
```

pymove.visualization.folium.**plot_event**(*move_data: DataFrame*, *event_lat: str = 'lat'*, *event_lon: str = 'lon'*, *event_point: str = 'purple'*, *radius: float = 2*, *base_map: Map | None = None*, *slice_tags: list | None = None*, *tiles: str = 'CartoDB positron'*, *save_as_html: bool = False*, *filename: str = 'events.html'*) → Map

Receives a MoveDataFrame and returns a folium map with events.

Parameters

- **move_data** (*DataFrame*) – Trajectory input data
- **event_lat** (*str, optional*) – Latitude column name, by default LATITUDE.
- **event_lon** (*str, optional*) – Longitude column name, by default LONGITUDE.
- **event_point** (*str, optional*) – Event color, by default EVENT_POI
- **radius** (*float, optional*) – radius size, by default 2.
- **base_map** (*Folium map, optional*) – A folium map to plot. If None a map. If None a map will be created, by default None.
- **tiles** (*str, optional, by default TILES[0]*) –
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'events.html'.

Returns

Return type A folium map.

Examples

```
>>> from pymove.visualization.folium import plot_event
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_event(move_df)
```

pymove.visualization.folium.**plot_markers**(*move_data: DataFrame*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *filename: str = 'markers.html'*) → Map

Generate visualization of Heat Map using folium plugin.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None

- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map, by default None
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map, by default None
- **zoom_start** (*float, optional*) – Initial zoom level for the map, by default 12
- **radius** (*float, optional*) – Radius of each "point" of the heatmap, by default 8
- **base_map** (*Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin and zoom_start, by default None
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'markers.html'

Returns folium Map

Return type Map

Examples

```
>>> from pymove.visualization.folium import plot_markers
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_markers(move_df)
```

pymove.visualization.folium.**plot_poi**(*move_data: DataFrame*, *poi_lat: str = 'lat'*, *poi_lon: str = 'lon'*, *poi_point: str = 'red'*, *radius: float = 2*, *base_map: Map | None = None*, *slice_tags: list | None = None*, *tiles: str = 'CartoDB positron'*, *save_as_html: bool = False*, *filename: str = 'pois.html'*) → Map

Receives a MoveDataFrame and returns a folium map with poi points.

Parameters

- **move_data** (*DataFrame*) – Trajectory input data
- **poi_lat** (*str, optional*) – Latitude column name, by default LATITUDE.
- **poi_lon** (*str, optional*) – Longitude column name, by default LONGITUDE.
- **poi_point** (*str, optional*) – Poi point color, by default POI_POINT.
- **radius** (*float, optional*) – radius size, by default 2.
- **base_map** (*Folium map, optional*) – A folium map to plot. If None a map. If None a map will be created, by default None.
- **slice_tags** (*optional, by default None.*) –
- **tiles** (*str, optional, by default TILES[0]*) – The map type.

- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'pois.html'.

Returns Represents a folium map with visualization.

Return type folium.folium.Map.

Examples

```
>>> from pymove.visualization.folium import plot_poi
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_poi(move_df)
```

pymove.visualization.folium.**plot_points**(*move_data: DataFrame, user_lat: str = 'lat', user_lon: str = 'lon', user_point: str = 'orange', radius: float = 2, base_map: Map | None = None, slice_tags: list | None = None, tiles: str = 'CartoDB positron', save_as_html: bool = False, filename: str = 'points.html'*) → Map

Generates a folium map with the trajectories plots and a point.

Parameters

- **move_data** (*Dataframe*) – Trajectory data.
- **user_lat** (*str, optional*) – Latitude column name, by default LATITUDE.
- **user_lon** (*str, optional*) – Longitude column name, by default LONGITUDE.
- **user_point** (*str, optional*) – The point color, by default USER_POINT.
- **radius** (*float, optional*) – radius size, by default 2.
- **sort** (*Boolean, optional*) – If True the data will be sorted, by default False.
- **base_map** (*Folium map, optional*) – A folium map to plot the trajectories. If None a map will be created, by default None.
- **slice_tags** (*optional, by default None.*) –
- **tiles** (*str, optional, by default TILES[0]*) – The map type.
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'points.html'.

Returns A folium map

Return type Map

Examples

```
>>> from pymove.visualization.folium import plot_points
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> plot_points(move_df)
```

pymove.visualization.folium.**plot_stops**(*move_data: PandasMoveDataFrame*, *radius: float = 0*, *weight: float = 3*, *id_: int | None = None*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *legend: bool = True*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *color: str | list[str] | None = None*, *filename: str = 'plot_stops.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **radius** (*float, optional*) – The radius value is used to determine if a segment is a stop. If the value of the point in target_label is greater than radius, the segment is a stop, by default 0
- **weight** (*float, optional*) – Stroke width in pixels, by default 3
- **id** (*int, optional*) – Trajectory id to plot, by default None
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_stops.html'.

Returns a folium map with visualization

Return type Map

Raises

- `KeyError` – If period value is not found in dataframe
- `IndexError` – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_stops
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> plot_stops(move_df)
```

pymove.visualization.folium.**plot_traj_timestamp_geo_json**(*move_data: pandas.core.frame.DataFrame*, *label_lat: str = 'lat'*, *label_lon: str = 'lon'*, *label_datetime: str = 'datetime'*, *tiles: str = 'CartoDB positron'*, *save_as_html: bool = False*, *filename: str = 'events.html'*) → folium.folium.Map

Plot trajectories wit geo_json.

Parameters

- **move_data** (*DataFrame.*) – Input trajectory data.
- **label_datetime** (*str, optional, by default DATETIME.*) – date_time column label.
- **label_lat** (*str, optional, by default LATITUDE.*) – latitude column label.
- **label_lon** (*str, optional, by default LONGITUDE.*) – longitude column label.
- **tiles** (*str, optional*) – map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'events.html'.

Returns A folium map.

Return type Map

Examples

```
>>> from pymove.visualization.folium import plot_traj_timestamp_geo_json
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_traj_timestamp_geo_json(move_df)
```

pymove.visualization.folium.**plot_trajectories**(*move_data: DataFrame*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *legend: bool = True*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *color: str | list[str] | None = None*, *color_by_id: dict | None = None*, *filename: str = 'plot_trajectories.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data.
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **color_by_id** (*dict, optional*) – A dictionary where the key is the trajectory id and value is a color(str), by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectory.html'.

Returns a folium map with visualization.

Return type Map

Examples

```
>>>  from pymove.visualization.folium import plot_trajectories
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_trajectories(move_df)
```

pymove.visualization.folium.**plot_trajectory_by_date**(*move_data: PandasMoveDataFrame, start_date: str | date, end_date: str | date, id_: int | None = None, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, legend: bool = True, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, color: str | list[str] | None = None, color_by_id: dict | None = None, filename: str = 'plot_trajectories_by_date.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **start_date** (*str*) – Represents start date of time period.
- **end_date** (*str*) – Represents end date of time period.
- **id** (*int, optional*) – Trajectory id to plot, by default None
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.

- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **color_by_id** (*dict, optional*) – A dictionary where the key is the trajectory id and value is a color, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectories_by_date.html'.

Returns a folium map with visualization

Return type Map

Raises

- `KeyError` – If period value is not found in dataframe
- `IndexError` – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_trajectory_by_date
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_trajectory_by_date(
>>>     move_df,
>>>     start_date='2008-10-23 05:53:05',
>>>     end_date='2008-10-23 23:43:56'
>>> )
```

pymove.visualization.folium.**plot_trajectory_by_day_week**(*move_data: PandasMoveDataFrame, day_week: str, id_: int | None = None, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, legend: bool = True, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, color: str | list[str] | None = None, color_by_id: dict | None = None, filename: str = 'plot_trajectories_by_day_week.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **day_week** (*str*) – Day of the week
- **id** (*int*) – Trajectory id to plot, by default None
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **color_by_id** (*dict, optional*) – A dictionary where the key is the trajectory id and value is a color, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectories_by_day_week.html'.

Returns a folium map with visualization

Return type Map

Raises

- `KeyError` – If period value is not found in dataframe
- `IndexError` – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_trajectory_by_day_week
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_trajectory_by_day_week(move_df, day_week='Friday')
```

pymove.visualization.folium.**plot_trajectory_by_hour**(*move_data: PandasMoveDataFrame*, *start_hour: str*, *end_hour: str*, *id_: int | None = None*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *legend: bool = True*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *color: str | list[str] | None = None*, *color_by_id: dict | None = None*, *filename: str = 'plot_trajectories_by_hour.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **start_hour** (*str*) – Represents start hour of time period.
- **end_hour** (*str*) – Represents end hour of time period.
- **id** (*int, optional*) – Trajectory id to plot, by default None
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **color_by_id** (*dict, optional*) – A dictionary where the key is the trajectory id and value is a color, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectories_by_hour.html'.

Returns a folium map with visualization

Return type Map

Raises

- `KeyError` – If period value is not found in dataframe
- `IndexError` – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_trajectory_by_hour
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    1
4   39.984217   116.319422  2008-10-23 05:53:21    1
>>> plot_trajectory_by_hour(move_df, start_hour=4, end_hour=6)
```

pymove.visualization.folium.**plot_trajectory_by_id**(*move_data: DataFrame*, *id_: int*, *n_rows: int | None = None*, *lat_origin: float | None = None*, *lon_origin: float | None = None*, *zoom_start: float = 12*, *legend: bool = True*, *base_map: Map | None = None*, *tile: str = 'CartoDB positron'*, *save_as_html: bool = False*, *color: str | list[str] | None = None*, *filename: str = 'plot_trajectories.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **id** (*int*) – Trajectory id to plot
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.

- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectory_by_id.html'.

Returns a folium map with visualization

Return type Map

Raises `IndexError` – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_trajectory_by_id
>>> move_df.head()
         lat          lon             datetime   id
0  39.984094  116.319236  2008-10-23 05:53:05    1
1  39.984198  116.319322  2008-10-23 05:53:06    1
2  39.984224  116.319402  2008-10-23 05:53:11    1
3  39.984211  116.319389  2008-10-23 05:53:16    2
4  39.984217  116.319422  2008-10-23 05:53:21    2
>>> plot_trajectory_by_id(move_df, id_=1)
```

pymove.visualization.folium.**plot_trajectory_by_period**(*move_data: PandasMoveDataFrame, period: str, id_: int | None = None, n_rows: int | None = None, lat_origin: float | None = None, lon_origin: float | None = None, zoom_start: float = 12, legend: bool = True, base_map: Map | None = None, tile: str = 'CartoDB positron', save_as_html: bool = False, color: str | list[str] | None = None, color_by_id: dict | None = None, filename: str = 'plot_trajectories_by_period.html'*) → Map

Generate visualization of all trajectories with folium.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **period** (*str*) – Period of the day
- **id** (*int*) – Trajectory id to plot, by default None
- **n_rows** (*int, optional*) – Represents number of data rows that are will plot, by default None.
- **lat_origin** (*float, optional*) – Represents the latitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.
- **lon_origin** (*float, optional*) – Represents the longitude which will be the center of the map. If not entered, the first data from the dataset is used, by default None.

- **zoom_start** (*int, optional*) – Initial zoom level for the map, by default 12.
- **legend** (*boolean*) – Whether to add a legend to the map, by default True
- **base_map** (*folium.folium.Map, optional*) – Represents the folium map. If not informed, a new map is generated using the function create_base_map(), with the lat_origin, lon_origin

  and zoom_start, by default None.
- **tile** (*str, optional*) – Represents the map tiles, by default TILES[0]
- **save_as_html** (*bool, optional*) – Represents if want save this visualization in a new file .html, by default False.
- **color** (*str, list, optional*) – Represents line colors of visualization. Can be a single color name, a list of colors or a colormap name, by default None.
- **color_by_id** (*dict, optional*) – A dictionary where the key is the trajectory id and value is a color, by default None.
- **filename** (*str, optional*) – Represents the file name of new file .html, by default 'plot_trajectories_by_period.html'.

Returns a folium map with visualization

Return type Map

Raises

- KeyError – If period value is not found in dataframe
- IndexError – If there is no user with the id passed

Examples

```
>>> from pymove.visualization.folium import plot_trajectory_by_period
>>> move_df.head()
          lat          lon              datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> plot_trajectory_by_period(move_df, period='Early morning')
```

pymove.visualization.folium.**save_map**(*move_data: DataFrame, filename: str, tiles: str = 'CartoDB positron', label_id: str = 'id', cmap: str = 'Set1', return_map: bool = False*) → Map | None

Save a visualization in a map in a new file.

Parameters

- **move_data** (*DataFrame*) – Input trajectory data
- **filename** (*Text*) – Represents the filename path
- **tiles** (*str, optional*) – Represents the type_ of tile that will be used on the map, by default TILES[0]
- **label_id** (*str, optional*) – Represents column name of trajectory id, by default TRAJ_ID

- **cmap** (*str, optional*) – Color map to use, by default 'Set1'
- **return_map** (*bool, optional*) – Represents the Colormap, by default False

Returns folium map or None

Return type Map

Examples

```
>>> from pymove.visualization.folium import save_map
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    1
4   39.984217   116.319422   2008-10-23 05:53:21    1
>>> save_map(df, filename='test.map')
```

pymove.visualization.matplotlib module

Matplolib operations.

show_object_id_by_date, plot_trajectories, plot_trajectory_by_id, plot_grid_polygons, plot_all_features plot_coords, plot_bounds, plot_line

pymove.visualization.matplotlib.**plot_all_features**(*move_data: DataFrame, dtype: Callable = <class 'float'>, figsize: tuple[float, float] = (21, 15), return_fig: bool = False, save_fig: bool = False, name: str = 'features.png'*) → figure | None

Generate a visualization for each columns that type is equal dtype.

Parameters

- **move_data** (*dataframe*) – Dataframe with trajectories
- **dtype** (*callable, optional*) – Represents column type, by default np.float64
- **figsize** (*tuple(float, float), optional*) – Represents dimensions of figure, by default (21, 15)
- **return_fig** (*bool, optional*) – Represents whether or not to return the generated picture, by default False
- **save_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False
- **name** (*str, optional*) – Represents name of a file, by default 'features.png'

Returns The generated picture or None

Return type figure

Raises AttributeError – If there are no columns with the specified type

Examples

```
>>>  from pymove.visualization.matplotlib import plot_all_features
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236  2008-10-23 05:53:05    1
1   39.984198   116.319322  2008-10-23 05:53:06    1
2   39.984224   116.319402  2008-10-23 05:53:11    1
3   39.984211   116.319389  2008-10-23 05:53:16    2
4   39.984217   116.319422  2008-10-23 05:53:21    2
>>>  plot_all_features(move_df)
```

pymove.visualization.matplotlib.**plot_bounds**(*ax: axes*, *ob: LineString | MultiLineString*, *color='b'*)

Plot the limits of geometric object.

Parameters

- **ax** (*axes*) – Single axes object
- **ob** (*LineString or MultiLineString*) – Geometric object formed by lines.
- **color** (*str, optional*) – Sets the geometric object color, by default 'b'

Example

```
>>> from pymove.visualization.matplotlib import plot_bounds
>>> import matplotlib.pyplot as plt
>>> bounds = LineString([(1, 1), (1, 2), (2, 2), (2, 3)])
>>> _, ax = plt.subplots(figsize=(21, 9))
>>> plot_bounds(ax, bounds)
```

pymove.visualization.matplotlib.**plot_coords**(*ax: matplotlib.pyplot.axes*, *ob: shapely.geometry.base.BaseGeometry*, *color: str = 'r'*)

Plot the coordinates of each point of the object in a 2D chart.

Parameters

- **ax** (*axes*) – Single axes object
- **ob** (*geometry object*) – Any geometric object
- **color** (*str, optional*) – Sets the geometric object color, by default 'r'

Example

```
>>> from pymove.visualization.matplotlib import plot_coords
>>> import matplotlib.pyplot as plt
>>> coords = LineString([(1, 1), (1, 2), (2, 2), (2, 3)])
>>> _, ax = plt.subplots(figsize=(21, 9))
>>> plot_coords(ax, coords)
```

pymove.visualization.matplotlib.**plot_grid_polygons**(*data: DataFrame*, *grid: Grid | None = None*, *markersize: float = 10*, *linewidth: float = 2*, *figsize: tuple[int, int] = (10, 10)*, *return_fig: bool = False*, *save_fig: bool = False*, *name: str = 'grid.png'*) → figure | None

Generate a visualization with grid polygons.

Parameters

- **data** (*DataFrame*) – Input trajectory data
- **markersize** (*float, optional*) – Represents visualization size marker, by default 10
- **linewidth** (*float, optional*) – Represents visualization size line, by default 2
- **figsize** (*tuple(int, int), optional*) –

  Represents the size (float: width, float: height) of a figure, by default (10, 10)
- **return_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False
- **save_fig** (*bool, optional*) – Wether to save the figure, by default False
- **name** (*str, optional*) – Represents name of a file, by default 'grid.png'

Returns The generated picture or None

Return type figure

Raises If the dataframe does not contains the POLYGON feature

IndexError If there is no user with the id passed

pymove.visualization.matplotlib.**plot_line**(*ax: matplotlib.pyplot.axes*, *ob: shapely.geometry.linestring.LineString*, *color: str = 'r'*, *alpha: float = 0.7*, *linewidth: float = 3*, *solid_capstyle: str = 'round'*, *zorder: float = 2*)

Plot a LineString.

Parameters

- **ax** (*axes*) – Single axes object
- **ob** (*LineString*) – Sequence of points.
- **color** (*str, optional*) – Sets the line color, by default 'r'
- **alpha** (*float, optional*) – Defines the opacity of the line, by default 0.7
- **linewidth** (*float, optional*) – Defines the line thickness, by default 3
- **solid_capstyle** (*str, optional*) – Defines the style of the ends of the line, by default 'round'
- **zorder** (*float, optional*) – Determines the default drawing order for the axes, by default 2

Example

```
>>> from pymove.visualization.matplotlib import plot_line
>>> import matplotlib.pyplot as plt
>>> line = LineString([(1, 1), (1, 2), (2, 2), (2, 3)])
>>> _, ax = plt.subplots(figsize=(21, 9))
>>> plot_line(ax, line)
```

pymove.visualization.matplotlib.**plot_trajectories**(*move_data: DataFrame*, *markers: str = 'o'*, *markersize: float = 12*, *figsize: tuple[float, float] = (10, 10)*, *return_fig: bool = False*, *save_fig: bool = False*, *name: str = 'trajectories.png'*) → figure | None

Generate a visualization that show trajectories.

Parameters

- **move_data** (*dataframe*) – Dataframe with trajectories
- **markers** (*str, optional*) – Represents visualization type marker, by default 'o'
- **markersize** (*float, optional*) – Represents visualization size marker, by default 12
- **figsize** (*tuple(float, float), optional*) – Represents dimensions of figure, by default (10, 10)
- **return_fig** (*bool, optional*) – Represents whether or not to return the generated picture, by default False
- **save_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False
- **name** (*str, optional*) – Represents name of a file, by default 'trajectories.png'

Returns The generated picture or None

Return type figure

Examples

```
>>>  from pymove.visualization.matplotlib import plot_trajectories
>>> move_df.head()
         lat          lon             datetime   id
0  39.984094  116.319236  2008-10-23 05:53:05    1
1  39.984198  116.319322  2008-10-23 05:53:06    1
2  39.984224  116.319402  2008-10-23 05:53:11    1
3  39.984211  116.319389  2008-10-23 05:53:16    2
4  39.984217  116.319422  2008-10-23 05:53:21    2
>>> plot_trajectories(move_df)
```

pymove.visualization.matplotlib.**plot_trajectory_by_id**(*move_data: DataFrame*, *id_: int | str*, *label: str = 'id'*, *feature: str | None = None*, *value: Any | None = None*, *linewidth: float = 3*, *markersize: float = 20*, *figsize: tuple[float, float] = (10, 10)*, *return_fig: bool = False*, *save_fig: bool = False*, *name: str | None = None*) → figure | None

Generate a visualization that shows a trajectory with the specified tid.

Parameters

- **move_data** (*dataframe*) – Dataframe with trajectories
- **id** (*int, str*) – Represents the trajectory tid
- **label** (*str, optional*) – Feature with trajectories tids, by default TID
- **feature** (*str, optional*) – Name of the feature to highlight on plot, by default None
- **value** (*any, optional*) – Value of the feature to be highlighted as green marker, by default None
- **linewidth** (*float, optional*) – Represents visualization size line, by default 2
- **markersize** (*float, optional*) – Represents visualization size marker, by default 20
- **figsize** (*tuple(float, float), optional*) – Represents dimensions of figure, by default (10, 10)
- **return_fig** (*bool, optional*) – Represents whether or not to return the generated picture, by default False
- **save_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False
- **name** (*str, optional*) – Represents name of a file, by default None

Returns Trajectory with the specified tid. The generated picture.

Return type PandasMoveDataFrame', figure

Raises

- `KeyError` – If the dataframe does not contains the TID feature
- `IndexError` – If there is no trajectory with the tid passed

Examples

```
>>> from pymove.visualization.matplotlib import  plot_traj_by_id
>>> move_df
          lat          lon             datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
```

(continues on next page)

(continued from previous page)

```
3   39.984211   116.319389   2008-10-23 05:53:16    2
4   39.984217   116.319422   2008-10-23 05:53:21    2
>>> plot_traj_by_id(move_df_3, 1, label='id)
>>> plot_traj_by_id(move_df_3, 2, label='id)
```

pymove.visualization.matplotlib.**show_object_id_by_date**(*move_data: 'PandasMoveDataFrame' | 'DaskMoveDataFrame'*, *kind: list | None = None*, *figsize: tuple[float, float] = (21, 9)*, *return_fig: bool = False*, *save_fig: bool = False*, *name: str = 'shot_points_by_date.png'*) → figure | None

Generates four visualizations based on datetime feature.

- Bar chart trajectories by day periods
- Bar chart trajectories day of the week
- Line chart trajectory by date
- Line chart of trajectory by hours of the day.

Parameters

- **move_data** (*pymove.core.MoveDataFrameAbstract subclass.*) – Input trajectory data.
- **kind** (*list, optional*) – Determines the kinds of each plot, by default None
- **figsize** (*tuple, optional*) – Represents dimensions of figure, by default (21,9).
- **return_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False.
- **save_fig** (*bool, optional*) – Represents whether or not to save the generated picture, by default False.
- **name** (*String, optional*) – Represents name of a file, by default 'shot_points_by_date.png'.

Returns The generated picture or None

Return type figure

References

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html

Examples

```
>>> from pymove.visualization.matplotlib import show_object_id_by_date
>>> move_df.head()
          lat          lon             datetime   id
0   39.984094   116.319236   2008-10-23 05:53:05    1
```

(continues on next page)

(continued from previous page)

```
1   39.984198   116.319322   2008-10-23 05:53:06    1
2   39.984224   116.319402   2008-10-23 05:53:11    1
3   39.984211   116.319389   2008-10-23 05:53:16    2
4   39.984217   116.319422   2008-10-23 05:53:21    2
>>> show_object_id_by_date(move_df)
```

Module contents

Contains functions to create visualizations of trajectories.

folium, matplotlib

4.1.1.2 Module contents

PyMove.

Provides processing and visualization of trajectories and other spatial-temporal data

4.1.2 example notebooks

4.1.2.1 Notebooks

00 - What is PyMove?

PyMove is a Python library, open-source, that have operations to handling trajectories data, ranging from data representation, preprocessing operations, models, and visualization techniques.

PyMove **proposes**: - A familiar and similar syntax to Pandas; - Clear documentation; - Extensibility, since you can implement your main data structure by manipulating other data structures such as Dask DataFrame, numpy arrays, etc., in addition to adding new modules; - Flexibility, as the user can switch between different data structures; - Operations for data preprocessing, pattern mining and data visualization.

Enviroment settings

1. Create an environment using **Conda**

```
conda create -n validacao-pymove python=x.x
```

2. Activate the environment

```
conda activate validacao-pymove
```

Using PyMove

1. Clone this repository

```
git clone https://github.com/InsightLab/PyMove
```

2. Make a branch developer

```
git branch developer
```

3. Switch to a new branch

```
git checkout developer
```

4. Make a pull of branch

```
git pull origin developer
```

5. Switch to folder PyMove

```
cd PyMove
```

6. Install in developer mode

```
make dev
```

7. Now, use this command to use PyMove!

```
import pymove
```

What can you do with PyMove?

With Pymove you can handling trajectories data with operations of: - Grid - Preprocessing: this including segmentation, compression, noise filter, stay point detection and map matching techniques. - Data Visualization: exploring differents techniques and channels to view your data!

---

01 - Exploring MoveDataFrame

To work with Pymove you need to import the data into our data structure: **MoveDataFrame**!

MoveDataFrame is an abstraction that instantiates a new data structure that manipulates the structure the user wants. This is done using the Factory Method design pattern. This structure allows the interface to be implemented using different representations and libraries that manipulate the data.

We have an interface that delimits the scope that new implementing classes should have. We currently have two concrete classes that implement this interface: **PandasMoveDataFrame** and **DaskMoveDataFrame** (under construction), which use Pandas and Dask respectively for data manipulation.

It works like this: The user instantiating a MoveDataFrame provides a flag telling which library they want to use for manipulating this data.

Now that we understand the concept and data structure of PyMove, **hands on!**

MoveDataFrame

A MoveDataFrame must contain the columns: - `lat`: represents the latitude of the point. - `lon`: represents the longitude of the point. - `datetime`: represents the date and time of the point.

In addition, the user can enter several other columns as trajectory id. **If the id is not entered, the points are supposed to belong to the same path**.

Creating a MoveDataFrame

A MoveDataFrame can be created by passing a Pandas DataFrame, a list, dict or even reading a file. Look:

```
import pymove as pm
from pymove import MoveDataFrame
```

From a list

```
list_data = [
    [39.984094, 116.319236, '2008-10-23 05:53:05', 1],
    [39.984198, 116.319322, '2008-10-23 05:53:06', 1],
    [39.984224, 116.319402, '2008-10-23 05:53:11', 1],
    [39.984224, 116.319402, '2008-10-23 05:53:11', 1],
    [39.984224, 116.319402, '2008-10-23 05:53:11', 1],
    [39.984224, 116.319402, '2008-10-23 05:53:11', 1]
]
move_df = MoveDataFrame(data=list_data, latitude="lat", longitude="lon", datetime=
↪"datetime", traj_id="id")
move_df.head()
```

From a dict

```
dict_data = {
    'lat': [39.984198, 39.984224, 39.984094],
    'lon': [116.319402, 116.319322, 116.319402],
    'datetime': ['2008-10-23 05:53:11', '2008-10-23 05:53:06', '2008-10-23 05:53:06']
}

move_df = MoveDataFrame(data=dict_data, latitude="lat", longitude="lon", datetime=
↪"datetime", traj_id="id")
move_df.head()
```

From a DataFrame Pandas

```
import pandas as pd

df = pd.read_csv('geolife_sample.csv', parse_dates=['datetime'])
move_df = MoveDataFrame(data=df, latitude="lat", longitude="lon", datetime="datetime")

move_df.head()
```

From a file

```
move_df = pm.read_csv('geolife_sample.csv')
move_df.head()
```

Cool, huh? The default flag is Pandas. Look that:

```
type(move_df)
```

```
pymove.core.pandas.PandasMoveDataFrame
```

Let's try creating one with Dask!

```
move_df = pm.read_csv('geolife_sample.csv', type_='dask')
move_df.head()
```

```
type(move_df)
```

```
pymove.core.dask.DaskMoveDataFrame
```

What's in MoveDataFrame?

The MoveDataFrame stores the following information:

```
orig_df = pm.read_csv('geolife_sample.csv')
move_df = orig_df.copy()
```

1. The kind of data he was instantiated

```
move_df.get_type()
```

```
'pandas'
```

```
move_df.columns
```

```
Index(['lat', 'lon', 'datetime', 'id'], dtype='object')
```

```
move_df.dtypes
```

```
lat                 float64
lon                 float64
datetime    datetime64[ns]
id                   int64
dtype: object
```

In addition to these attributes, we have some functions that allow us to:

1. View trajectory information

```
move_df.show_trajectories_info()
```

```
====================== INFORMATION ABOUT DATASET ======================

Number of Points: 217653

```

(continues on next page)

(continued from previous page)

```
Number of IDs objects: 2

Start Date:2008-10-23 05:53:05     End Date:2009-03-19 05:46:37

Bounding Box:(22.147577, 113.548843, 41.132062, 121.156224)


======================================================================
```

2. View the number of users

```
move_df.get_users_number()
```

```
1
```

3. Transform our data to

a. Numpy

```
move_df.to_numpy()
```

```
array([[39.984094, 116.319236, Timestamp('2008-10-23 05:53:05'), 1],
       [39.984198, 116.319322, Timestamp('2008-10-23 05:53:06'), 1],
       [39.984224, 116.319402, Timestamp('2008-10-23 05:53:11'), 1],
       ...,
       [39.999945, 116.327394, Timestamp('2009-03-19 05:46:12'), 5],
       [40.000015, 116.327433, Timestamp('2009-03-19 05:46:17'), 5],
       [39.999978, 116.32746, Timestamp('2009-03-19 05:46:37'), 5]],
      dtype=object)
```

b. Dicts

```
dict_data = move_df.to_dict()
dict_data.keys()
```

```
dict_keys(['lat', 'lon', 'datetime', 'id'])
```

c. DataFrames

```
df = move_df.to_data_frame()
print(type(move_df))
print(type(df))
df
```

```
<class 'pymove.core.pandas.PandasMoveDataFrame'>
<class 'pandas.core.frame.DataFrame'>
```

4. And even switch from a Pandas to Dask and back again!

```
new_move = move_df.convert_to('dask')
print(type(new_move))
move_df = new_move.convert_to('pandas')
print(type(move_df))
```

```
<class 'pymove.core.dask.DaskMoveDataFrame'>
<class 'pymove.core.pandas.PandasMoveDataFrame'>
```

5. You can also write files with

```
move_df.write_file('move_df_write_file.txt')
```

```
move_df.to_csv('move_data.csv')
```

6. Create a virtual grid

```
move_df.to_grid(8)
```

```
lon_min_x: 113.548843
lat_min_y: 22.147577
grid_size_lat_y: 263013
grid_size_lon_x: 105394
cell_size_by_degree: 7.218082747158498e-05
```

7. View the information of the last MoveDataFrame operation: operation name, operation time and memory use

```
move_df.last_operation
```

```
{'name': 'to_grid', 'time in seconds': 0.010914802551269531, 'memory': '0.0 B'}
```

8. Get data bound box

```
move_df.get_bbox()
```

```
(22.147577, 113.548843, 41.132062, 121.156224)
```

9. Create new columns:

a. `tid`: trajectory id based on Id and datetime

```
move_df.generate_tid_based_on_id_datetime()
move_df.head()
```

b. `date`: extract date on datetime

```
move_df.generate_date_features()
move_df.head()
```

c. `hour`: extract hour on datetime

```
move_df.generate_hour_features()
move_df.head()
```

d. `day`: day of the week from datatime.

```
move_df.generate_day_of_the_week_features()
move_df.head()
```

e. `period`: time of day or period from datatime.

```
move_df.generate_time_of_day_features()
move_df.head()
```

f. `dist_to_prev`, `time_to_prev`, `speed_to_prev`: create features of distance, time and speed to an GPS point P (lat, lon).

```
move_df = orig_df.copy()
move_df.generate_dist_time_speed_features()
move_df.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

g. `dist_to_prev`, `dist_to_next`, `dist_prev_to_next` : three distance in meters to an GPS point P (lat, lon).

```
move_df = orig_df.copy()
move_df.generate_dist_features()
move_df.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

h. `time_to_prev`, `time_to_next`, `time_prev_to_next` : three time in seconds to an GPS point P (lat, lon).

```
move_df = orig_df.copy()
move_df.generate_time_features()
move_df.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

i. `speed_to_prev`, `speed_to_next`, `speed_prev_to_next` : three speed in meters by seconds to an GPS point P (lat, lon).

```
move_df = orig_df.copy()
move_df.generate_speed_features()
move_df.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

j. `dist_to_prev`, `time_to_prev`, `speed_to_prev` : distance, time and speed from previous ponint

```
move_df = orig_df.copy()
move_df.generate_dist_time_speed_features(inplace=False)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

K. `situation`: column with move and stop points by radius.

```
move_df = orig_df.copy()
move_df.generate_move_and_stop_by_radius()
move_df.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

9. Get time difference between max and min datetime in trajectory data.

```
move_df.time_interval()
```

```
Timedelta('146 days 23:53:32')
```

And that's it! See upcoming notebooks to learn more about what PyMove can do!

02 - Exploring Preprocessing

Data preprocessing is a set of activities performed to prepare data for future analysis and data mining activities.

Load data from file

The dataset used in this tutorial is GeoLife GPS Trajectories. Available in https://www.microsoft.com/en-us/download/details.aspx?id=52367

```
from pymove import read_csv
```

```
df_move = read_csv('geolife_sample.csv')
```

```
df_move.show_trajectories_info()
df_move.head()
```

```
====================== INFORMATION ABOUT DATASET ======================

Number of Points: 217653

Number of IDs objects: 2

Start Date:2008-10-23 05:53:05     End Date:2009-03-19 05:46:37

Bounding Box:(22.147577, 113.548843, 41.132062, 121.156224)


======================================================================
```

Filtering

The filters module provides functions to perform different types of data filtering.

Importing the module:

```
from pymove import filters
```

A bounding box (usually shortened to bbox) is an area defined by two longitudes and two latitudes. The function by_bbox, filters points of the trajectories according to a chosen bounding box.

```
bbox = (22.147577, 113.54884299999999, 41.132062, 121.156224)
filt_df = filters.by_bbox(df_move, bbox)
filt_df.head()
```

by_datetime function filters point trajectories according to the time specified by the parameters: start_datetime and end_datetime.

```
filters.by_datetime(df_move, start_datetime = "2009-03-19 05:45:37", end_datetime =
↪"2009-03-19 05:46:17")
```

by label function filters trajectories points according to specified value and column label, set by value and label_name respectively.

```
filters.by_label(df_move, value = 116.327219, label_name = "lon").head()
```

by_id function filters trajectories points according to selected trajectory id.

```
filters.by_id(df_move, id_=5).head()
```

A tid is the result of concatenation between the id and date of a trajectory. The by_tid function filters trajectory points according to the tid specified by the tid_ parameter.

```
df_move.generate_tid_based_on_id_datetime()
filters.by_tid(df_move, "12008102305").head()
```

clean_consecutive_duplicates function removes consecutives duplicate rows of the Dataframe. Optionally only certaind columns can be consider, this is defined by the parameter subset, in this example only the lat column is considered.

```
filtered_df = filters.clean_consecutive_duplicates(df_move, subset = ["lat"])
len(filtered_df)
```

```
196142
```

clean_gps_jumps_by_distance function removes from the dataframe the trajectories points that are outliers.

```
filters.clean_gps_jumps_by_distance(df_move)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

clean_gps_nearby_points_by_distances function removes points from the trajectories when the distance between them and the point before is smaller than the parameter radius_area.

```
filters.clean_gps_nearby_points_by_distances(df_move, radius_area=10)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

clean_gps_nearby_points_by_speed function removes points from the trajectories when the speed of travel between them and the point before is smaller than the value set by the parameter speed_radius.

```
filters.clean_gps_nearby_points_by_speed(df_move, speed_radius=40.0)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

clean_gps_speed_max_radius function recursively removes trajectories points with speed higher than the value set by the user.

```
filters.clean_gps_speed_max_radius(df_move)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

clean_trajectories_with_few_points function removes from the given dataframe, trajectories with fewer points than was specified by the parameter min_points_per_trajectory.

```
filters.clean_trajectories_with_few_points(df_move)
```

Segmentation

The segmentation module are used to segment trajectories based on different parameters.

Importing the module:

```
from pymove import segmentation
```

bbox_split function splits the bounding box in grids of the same size. The number of grids is defined by the parameter number_grids.

```
bbox = (22.147577, 113.54884299999999, 41.132062, 121.156224)
segmentation.bbox_split(bbox, number_grids=4)
```

by_dist_time_speed functions segments the trajectories into clusters based on distance, time and speed.

```
segmentation.by_dist_time_speed(
    df_move,
    max_dist_between_adj_points=5000,
    max_time_between_adj_points=800,
    max_speed_between_adj_points=60.0
)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

by_max_dist function segments the trajectories into clusters based on distance.

```
segmentation.by_max_dist(df_move, max_dist_between_adj_points=4000)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

by_max_time function segments the trajectories into clusters based on time.

```
segmentation.by_max_time(df_move, max_time_between_adj_points=1000)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

by_max_speed function segments the trajectories into clusters based on speed.

```
segmentation.by_max_speed(df_move, max_speed_between_adj_points=70.0)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

Stay point detection

A stay point is location where a moving object has stayed for a while within a certain distance threshold. A stay point could stand different places such: a restaurant, a school, a work place.

Importing the module:

```
from pymove import stay_point_detection
```

create_or_update_move_stop_by_dist_time function creates or updates the stay points of the trajectories, based on distance and time metrics.

```
stay_point_detection.create_or_update_move_stop_by_dist_time(df_move, dist_radius=40,␣
↪time_radius=1000)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=3512)))
```

create_or_update_move_and_stop_by_radius function creates or updates the stay points of the trajectories, based on distance.

```
stay_point_detection.create_or_update_move_and_stop_by_radius(df_move, radius=2)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

Compression

Importing the module:

```
from pymove import compression
```

The function below is used to reduce the size of the trajectory, the stop points are used to make the compression.

```
df_compressed = compression.compress_segment_stop_to_point(df_move)
len(df_move), len(df_compressed)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=4809)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=285)))
```

```
(217653, 65620)
```

03 - Exploring Visualization

PyMove also has a visualization module to perform visual analysis on the data. These views make use of libraries like Matplotlib and Folium!

Hands-on!

1. Imports

```
import pymove as pm
from pymove.visualization import folium as f, matplotlib as mpl
from pymove.utils import visual
```

2. Load Data

```
move_df = pm.read_csv('geolife_sample.csv')
chunk1 = move_df[move_df['id'] == 1].head(5000)
chunk5 = move_df[move_df['id'] == 5].head(5000)
move_df = chunk1.append(chunk5)
move_df.head()
```

3. Exploring visualization module

a. Generate colors

We have a function that allows you to generate random colors so you can have more color options in your visualizations!

```
visual.generate_color()
```

```
'#00FF00'
```

Or passing intensity of each color will generate the color rgb

```
visual.rgb([0.6, 0.2, 0.2])
```

```
(51, 51, 153)
```

With this rbg tuple you can generate hex colors!

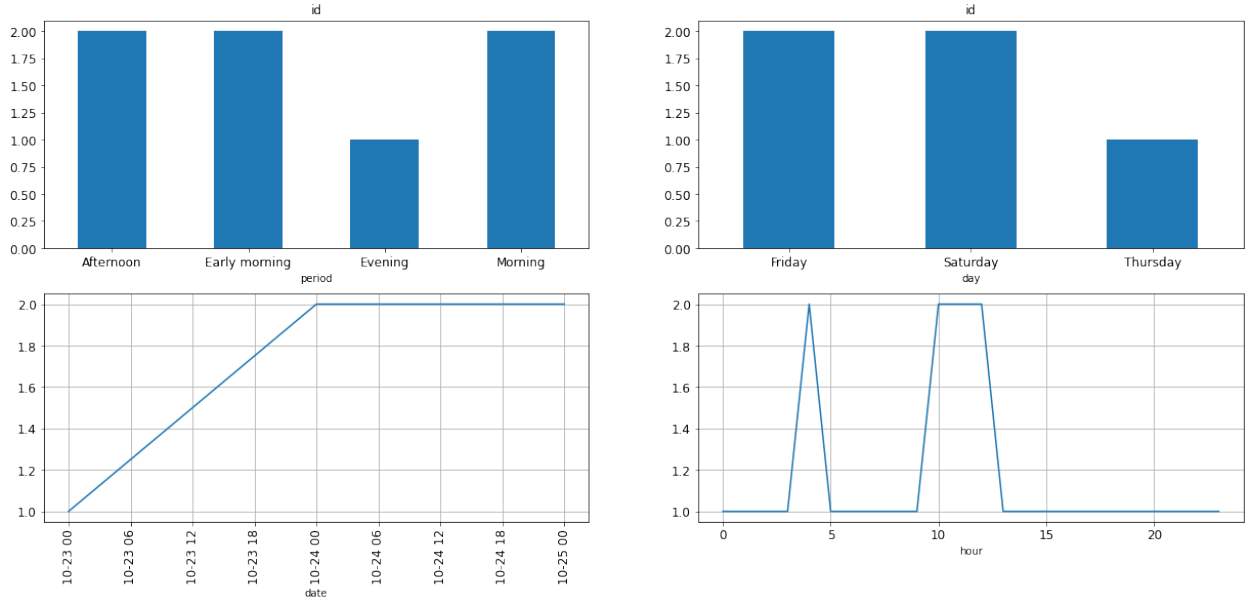
```
visual.hex_rgb([0.6, 0.2, 0.2])
```

```
'#333399'
```

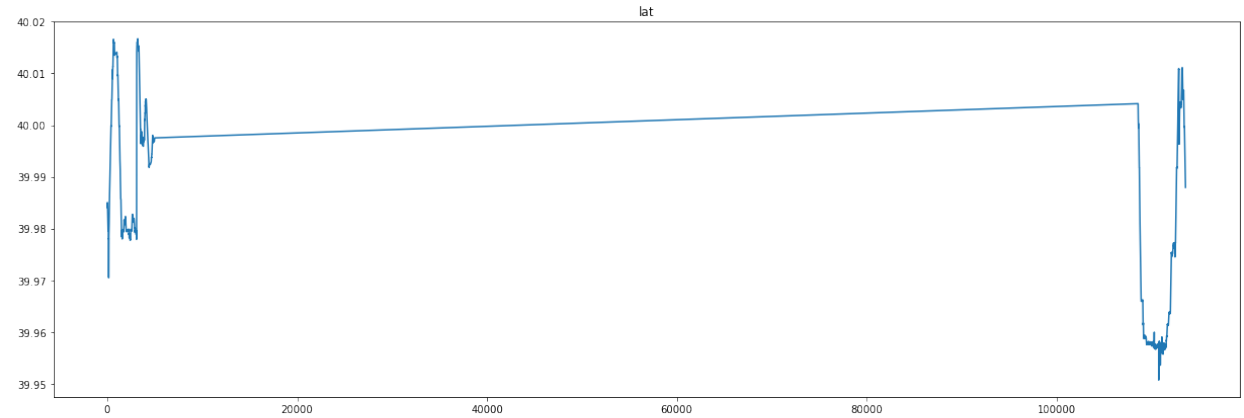
b. Exploring data over time

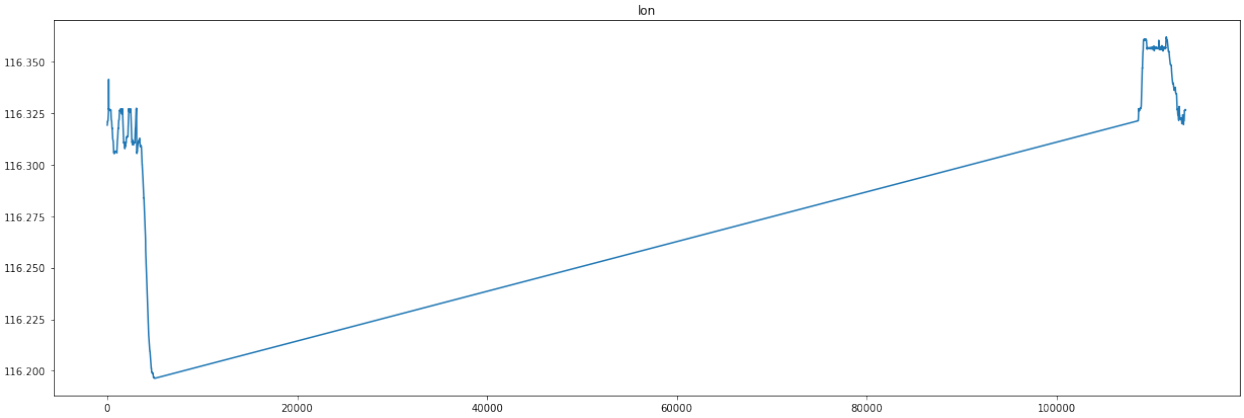
We can generate an overview that contains the distribution of data by time period, time, date, and day of the week to better understand how data is distributed.

```
mpl.show_object_id_by_date(move_df, return_fig=False)
```




```
mpl.plot_all_features(move_df, return_fig=False)
```





We can generate a visualization of the trajectory points filtered by: - **Day week**

```
f.plot_trajectory_by_day_week(move_df, 'Saturday')
```

- **Period of day**

```
f.plot_trajectory_by_period(move_df, 'Morning')
```

- **Period of time with start date and end date**

```
f.plot_trajectory_by_date(move_df, '2008-10-23', '2008-10-23')
```

- **Period of time with start and end time**

```
f.plot_trajectory_by_hour(move_df, 1, 3)
```

c. Exploring trajectories

- **Plot all trajectories**

```
f.plot_trajectories(move_df)
```

- **Plot trajectory by id**

```
f.plot_trajectory_by_id(move_df, 1, color="orange")
```

- **Heat map**

```
f.heatmap(move_df)
```

- **Heat map with time**

```
f.heatmap_with_time(move_df)
```

- **Plot cluster**

```
f.cluster(move_df, 1000)
```

- **Faster MarkerCluster**

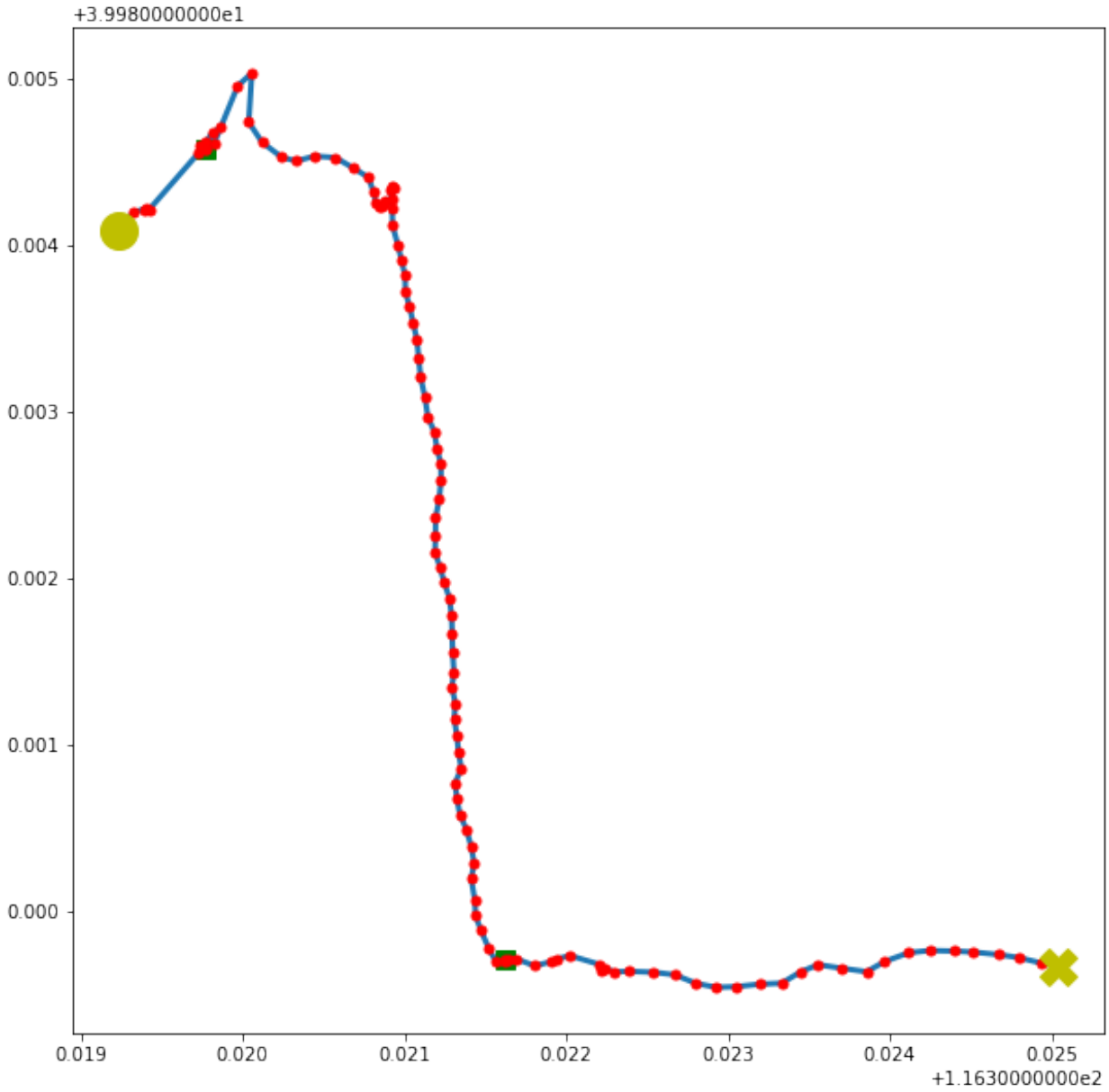
```
f.faster_cluster(move_df)
```

- **Plot stops points**

```
move_df.generate_tid_based_on_id_datetime()
move_df.generate_move_and_stop_by_radius()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

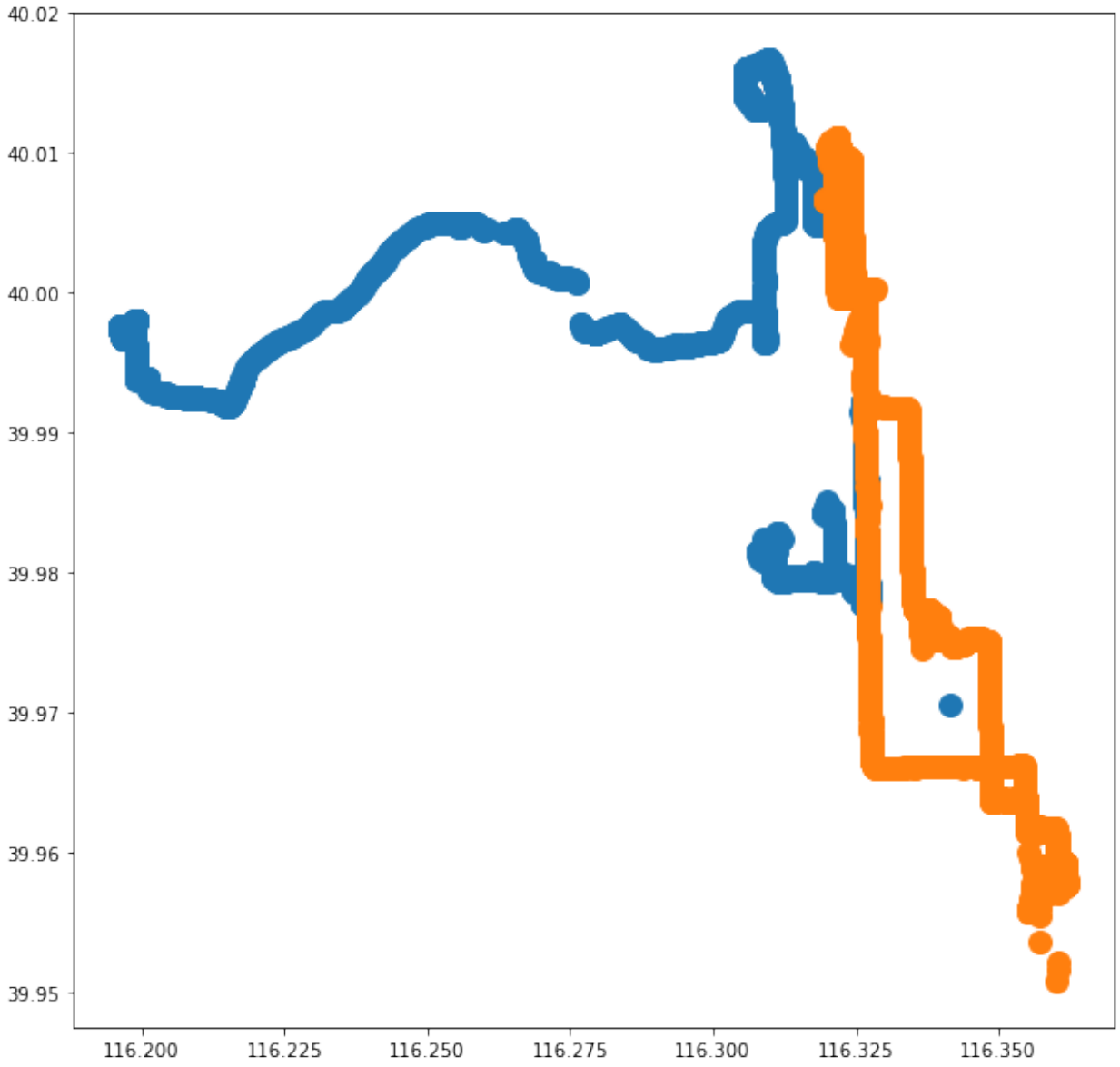
```
mpl.plot_trajectory_by_id(move_df, id_="12008102305", label='tid', feature="situation
↪", value="stop", return_fig=False)
```




```
f.plot_stops(move_df, n_rows=2000)
```

- **Show lat and lon GPS**

```
mpl.plot_trajectories(move_df, return_fig=False)
```




```
f.plot_markers(move_df, n_rows=400, zoom_start=13.5)
```

04 - Exploring Grid

In trajectories data mining process, there is a need frequent acess differents segments and samples of trajectories. With big volume data, those acesses can need time and processing. With this, it's necessary to adopt effective techniques to management and handling of this data, allowed fast recovery of data.

One approach to this technique takes geographic space into account, dividing it into grids, the so-called **grids**, and **creating a time index for the trajectories that fall into each cell of this grid**. **Each segment** that falls into a grid **is represented by a point with coordinates equal to the start time point and the end time point of the segment**.

In PyMove, grids are delimited by coordinates in a cartesian plan based tracing in bound box of data. Grids are represented by objects that have those attributes:

- **lon_min_x:** minimum longitude.

- **lat_min_y:** minimum latitude.
- **grid_size_lat_y:** grid latitude size.
- **grid_size_lon_x:** grid longitude size.
- **cell_size_by_degree:** cell size of Grid.

Imports

```
from pymove import read_csv
from pymove.core.grid import Grid
```

Load data

```
df = read_csv('geolife_sample.csv', parse_dates=['datetime'])
data = df[:1000]
data
```

Create virtual Grid

```
grid = Grid(data, 15)
```

```
grid.get_grid()
```

```
{'lon_min_x': 116.305468,
 'lat_min_y': 39.970511,
 'grid_size_lat_y': 341,
 'grid_size_lon_x': 266,
 'cell_size_by_degree': 0.00013533905150922183}
```

Create one polygon to point on grid

```
print(grid.create_one_polygon_to_point_on_grid(2, 1))
```

```
POLYGON ((116.3056033390515 39.97078167810302, 116.3056033390515 39.97091701715453,␣
↪116.305738678103 39.97091701715453, 116.305738678103 39.97078167810302, 116.
↪3056033390515 39.97078167810302))
```

Create or update index grid feature

```
grid.create_update_index_grid_feature(data)
```

```
data.head()
```

Create all polygons to all point on grid

```
grid_data = grid.create_all_polygons_to_all_point_on_grid(data)
```

```
grid_data.head()
```

Create all polygons on grid

```
grid.create_all_polygons_on_grid()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=341)))
```

```
(grid.grid_polygon).shape
```

```
(341, 266)
```

Get point to index grid

```
grid.point_to_index_grid(39.984094, 116.319236)
```

```
(100.0, 101.0)
```

Save grid to .pkl

```
grid.save_grid_pkl('teste.pkl')
```

Read .pkl to grid

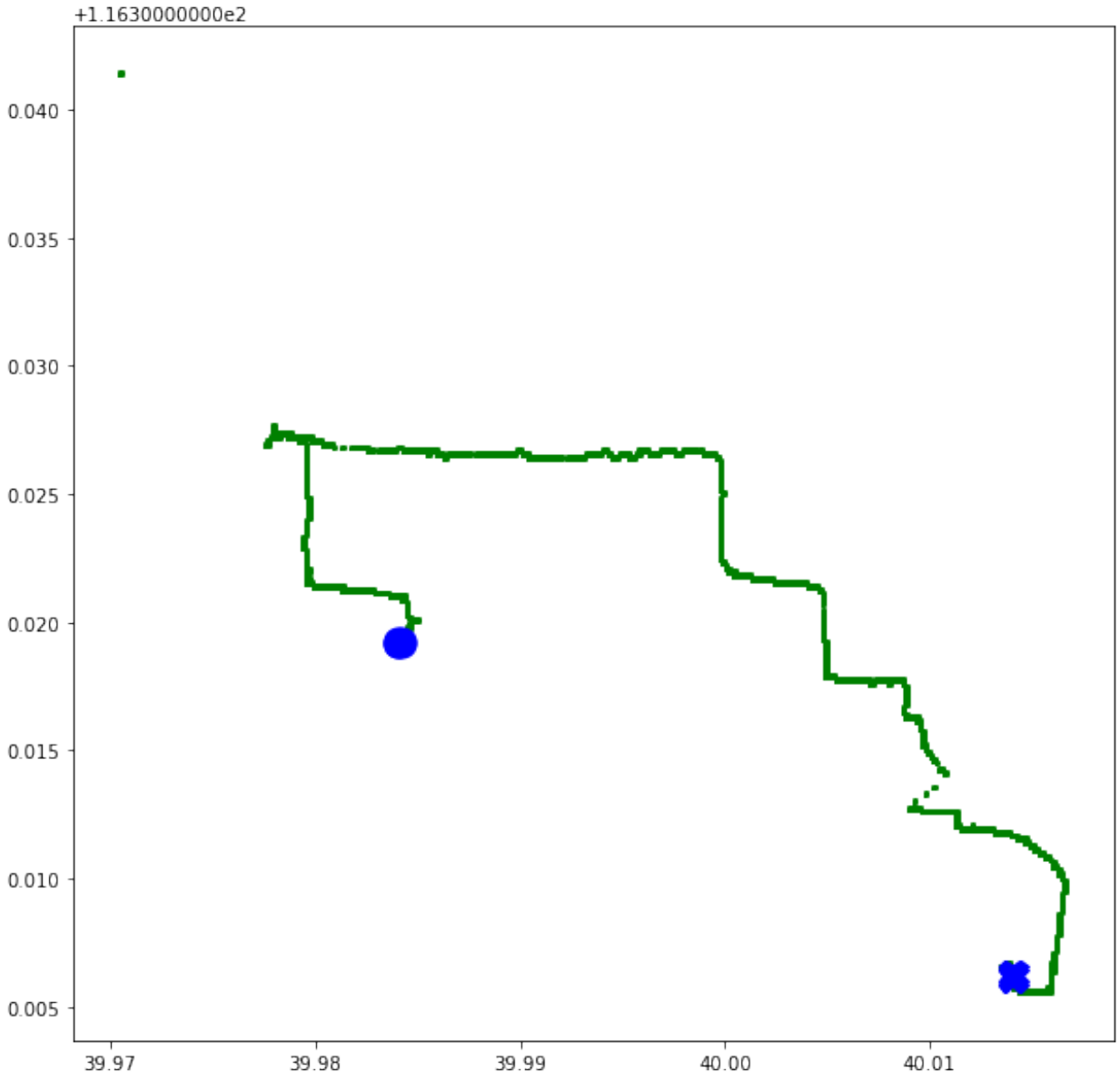
```
grid.read_grid_pkl('teste.pkl').get_grid()
```

```
{'lon_min_x': 116.305468,
 'lat_min_y': 39.970511,
 'grid_size_lat_y': 341,
 'grid_size_lon_x': 266,
 'cell_size_by_degree': 0.00013533905150922183}
```

Show a grid polygons

```
from pymove.visualization.matplotlib import plot_grid_polygons
```

```
plot_grid_polygons(grid_data, return_fig=False)
```



05 - Exploring Utils

Falar sobre para se trabalhar com trajetórias pode ser necessária algumas c onversões envolvendo tempo e data, distância e etc, fora outros utilitários.

Falar dos módulos presentes no pacote utils - constants - conversions - datetime - distances - math - trajectories - log - mem

Imports

```
import pymove.utils as utils
import pymove as pm
import datetime
```

Conversions

To transform latitude degree to meters, you can use function **lat_meters**. For example, you can convert Fortaleza's latitude -3.8162973555:

```
utils.conversions.lat_meters(-3.8162973555)
```

```
110826.6722516857
```

To concatenates list elements, joining them by the separator specified by the parameter "delimiter", you can use **list_to_str**

```
utils.conversions.list_to_str(["a", "b", "c", "d"], "-")
```

```
'a-b-c-d'
```

To concatenates the elements of the list, joining them by ",", , you can use **list_to_csv_str**

```
utils.conversions.list_to_csv_str(["a", "b", "c", "d"])
```

```
'a,b,c,d'
```

To concatenates list elements in consecutive element pairs, you can use **list_to_svm_line**

```
utils.conversions.list_to_svm_line(["a", "b", "c", "d"])
```

```
'a 1:b 2:c 3:d'
```

To convert longitude to X EPSG:3857 WGS 84/Pseudo-Mercator, you can use **lon_to_x_spherical**

```
utils.conversions.lon_to_x_spherical(-38.501597)
```

```
-4285978.172767829
```

To convert latitude to Y EPSG:3857 WGS 84/Pseudo-Mercator, you can use **lat_to_y_spherical**

```
utils.conversions.lat_to_y_spherical(-3.797864)
```

```
-423086.2213610324
```

To convert X EPSG:3857 WGS 84/Pseudo-Mercator to longitude, you can use **x_to_lon_spherical**

```
utils.conversions.x_to_lon_spherical(-4285978.172767829)
```

```
-38.501597000000004
```

To convert Y EPSG:3857 WGS 84/Pseudo-Mercator to latitude, you can use **y_to_lat_spherical**

```
utils.conversions.y_to_lat_spherical(-423086.2213610324)
```

```
-3.7978639999999944
```

```
move_data = pm.read_csv("geolife_sample.csv")
move_data.generate_dist_time_speed_features()
move_data.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

To convert values, in ms, in label_speed column to kmh, you can use **ms_to_kmh**

```
utils.conversions.ms_to_kmh(move_data, inplace=True)
move_data.head()
```

To convert values, in kmh, in label_speed column to ms, you can use **kmh_to_ms**

```
utils.conversions.kmh_to_ms(move_data, inplace=True)
move_data.head()
```

To convert values, in meters, in label_distance column to kilometer, you can use **meters_to_kilometers**

```
utils.conversions.meters_to_kilometers(move_data, inplace=True)
move_data.head()
```

To convert values, in kilometers, in label_distance column to meters, you can use **kilometers_to_meters**

```
utils.conversions.kilometers_to_meters(move_data, inplace=True)
move_data.head()
```

To convert values, in seconds, in label_distance column to minutes, you can use **seconds_to_minutes**

```
utils.conversions.seconds_to_minutes(move_data, inplace=True)
move_data.head()
```

To convert values, in minutes, in label_distance column to seconds, you can use **minute_to_seconds**

```
utils.conversions.minute_to_seconds(move_data, inplace=True)
move_data.head()
```

To convert in minutes, in label_distance column to hours, you can use **minute_to_hours**

```
utils.conversions.seconds_to_minutes(move_data, inplace=True)
utils.conversions.minute_to_hours(move_data, inplace=True)
move_data.head()
```

To convert in hours, in label_distance column to minute, you can use **hours_to_minutes**

```
utils.conversions.hours_to_minute(move_data, inplace=True)
move_data.head()
```

To convert in seconds, in label_distance column to hours, you can use **seconds_to_hours**

```
utils.conversions.minute_to_seconds(move_data, inplace=True)
utils.conversions.seconds_to_hours(move_data, inplace=True)
move_data.head()
```

To convert in seconds, in label_distance column to hours, you can use **hours_to_seconds**

```
utils.conversions.hours_to_seconds(move_data, inplace=True)
move_data.head()
```

Datetime

To converts a datetime in string“s format”%Y-%m-%d” or “%Y-%m-%d %H:%M:%S” to datetime”s format, you can use **str_to_datetime**.

```
utils.datetime.str_to_datetime('2018-06-29 08:15:27')
```

```
datetime.datetime(2018, 6, 29, 8, 15, 27)
```

To get date, in string’s format, from timestamp, you can use **date_to_str**.

```
utils.datetime.date_to_str(utils.datetime.str_to_datetime('2018-06-29 08:15:27'))
```

```
'2018-06-29'
```

To converts a datetime to an int representation in minutes, you can use **to_min**.

```
utils.datetime.datetime_to_min(datetime.datetime(2018, 6, 29, 8, 15, 27))
```

```
25504335
```

To do the reverse use: **min_to_datetime**

```
utils.datetime.min_to_datetime(25504335)
```

```
datetime.datetime(2018, 6, 29, 8, 15)
```

To get day of week of a date, you can use **to_day_of_week_int**, where 0 represents Monday and 6 is Sunday.

```
utils.datetime.to_day_of_week_int(datetime.datetime(2018, 6, 29, 8, 15, 27))
```

```
4
```

To indices if a day specified by the user is a working day, you can use **working_day**.

```
utils.datetime.working_day(datetime.datetime(2018, 6, 29, 8, 15, 27), country='BR')
```

```
True
```

```
utils.datetime.working_day(datetime.datetime(2018, 4, 21, 8, 15, 27), country='BR')
```

```
False
```

To get datetime of now, you can use **now_str**.

```
utils.datetime.now_str()
```

```
'2021-07-13 19:56:01'
```

To convert time in a format appropriate of time, you can use **deltatime_str**.

```
utils.datetime.deltatime_str(1082.7180936336517)
```

```
'18m:02.72s'
```

To converts a local datetime to a POSIX timestamp in milliseconds, you can use **timestamp_to_millis**.

```
utils.datetime.timestamp_to_millis("2015-12-12 08:00:00.123000")
```

```
1449907200123
```

To converts milliseconds to timestamp, you can use **millis_to_timestamp**.

```
utils.datetime.millis_to_timestamp(1449907200123)
```

```
Timestamp('2015-12-12 08:00:00.123000')
```

To get time, in string's format, from timestamp, you can use **time_to_str**.

```
utils.datetime.time_to_str(datetime.datetime(2018, 6, 29, 8, 15, 27))
```

```
'08:15:27'
```

To converts a time in string's format "%H:%M:%S" to datetime's format, you can use **str_to_time**.

```
utils.datetime.str_to_time("08:00:00")
```

```
datetime.datetime(1900, 1, 1, 8, 0)
```

To computes the elapsed time from a specific start time to the moment the function is called, you can use **elapsed_time_dt**.

```
utils.datetime.elapsed_time_dt(utils.datetime.str_to_time("08:00:00"))
```

```
3835166163375
```

To computes the elapsed time from the start time to the end time specifed by the user, you can use **diff_time**.

```
utils.datetime.diff_time(utils.datetime.str_to_time("08:00:00"), utils.datetime.str_
↪to_time("12:00:00"))
```

```
14400000
```

Distances

To calculate the great circle distance between two points on the earth, you can use **haversine**.

```
utils.distances.haversine(-3.797864,-38.501597,-3.797890, -38.501681)
```

```
9.757976024363016
```

To calculate the euclidean distance between two points on the earth, you can use **euclidean_distance_in_meters**.

```
utils.distances.euclidean_distance_in_meters(-3.797864,-38.501597,-3.797890, -38.
↪501681)
```

```
9.790407710249447
```

Math

To compute standard deviation, you can use **std**.

```
utils.math.std([600, 20, 5])
```

```
277.0178494048513
```

To compute the average of standard deviation, you can use **avg_std**.

```
utils.math.avg_std([600, 20, 5])
```

```
(208.33333333333334, 277.0178494048513)
```

To compute the standard deviation of sample, you can use **std_sample**.

```
utils.math.std_sample([600, 20, 5])
```

```
339.27619034251916
```

To compute the average of standard deviation of sample, you can use **avg_std_sample**.

```
utils.math.avg_std_sample([600, 20, 5])
```

```
(208.33333333333334, 339.27619034251916)
```

To computes the sum of the elements of the array, you can use **array_sum**.

To computes the sum of all the elements in the array, the sum of the square of each element and the number of elements of the array, you can use **array_stats**.

```
utils.math.array_stats([600, 20, 5])
```

```
(625.0, 360425.0, 3)
```

To perfomers interpolation and extrapolation, you can use **interpolation**.

```
utils.math.interpolation(15, 20, 65, 86, 5)
```

```
6.799999999999999
```

Trajectories

To read a csv file into a MoveDataFrame

```
move_data = utils.trajectories.read_csv('geolife_sample.csv')
type(move_data)
```

```
pymove.core.pandas.PandasMoveDataFrame
```

To invert the keys values of a dictionary

```
utils.trajectories.invert_dict({1: 'a', 2: 'b'})
```

```
{'a': 1, 'b': 2}
```

To flatten a nested dictionary

```
utils.trajectories.flatten_dict({'1': 'a', '2': {'3': 'b', '4': 'c'}})
```

```
{'1': 'a', '2_3': 'b', '2_4': 'c'}
```

To flatten a dataframe with dict as row values

```
df = move_data.head(3)
df['dict_column'] = [{'a': 1}, {'b': 2}, {'c': 3}]
df
```

```
utils.trajectories.flatten_columns(df, columns='dict_column')
```

To shift a sequence

```
utils.trajectories.shift([1., 2., 3., 4.], 1)
```

```
array([nan,  1.,  2.,  3.])
```

To fill a sequence with values from another

```
l1 = ['a', 'b', 'c', 'd', 'e']
utils.trajectories.fill_list_with_new_values(l1, [1, 2, 3])
l1
```

```
[1, 2, 3, 'd', 'e']
```

To transform a string representation back into a list

```
utils.trajectories.object_for_array('[1,2,3,4,5]')
```

```
array([1., 2., 3., 4., 5.], dtype=float32)
```

To convert a column with string representation back into a list

```
df['list_column'] = ['[1,2]', '[3,4]', '[5,6]']
```

```
df
```

```
utils.trajectories.column_to_array(df, column='list_column')
```

Log

```
mdf = pm.read_csv('geolife_sample.csv')
```

To cotrol the verbosity of pymove functions, use the logger

To change verbosity use the `utils.log.set_verbosity` method, or create and environment variable named `PYMOVE_VERBOSITY`

By default, the berbosity level is set to `INFO`

```
utils.log.logger
```

```
<Logger pymove (INFO)>
```

`INFO` shows only useful information, like progress bars

```
mdf.generate_dist_features(inplace=False).head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

`DEBUG` shows information from various steps in the functions

```
utils.log.set_verbosity('DEBUG')
mdf.generate_dist_features(inplace=False).head()
```

```
...Sorting by id and datetime to increase performance

...Set id as index to a higher performance


Creating or updating distance features in meters...
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

`WARN` hides all output except warnings and errors

```
utils.log.set_verbosity('WARN')
mdf.generate_dist_features(inplace=False).head()
```

Mem

```
utils.log.set_verbosity('INFO')
```

Calculate size of variable

```
utils.mem.total_size(mdf, verbose=True)
```

```
Size in bytes: 6965040, Type: <class 'pymove.core.pandas.PandasMoveDataFrame'>
```

```
6965040
```

Reduce size of dataframe

```
utils.mem.reduce_mem_usage_automatic(mdf)
```

```
Memory usage of dataframe is 6.64 MB
Memory usage after optimization is: 2.70 MB
Decreased by 59.4 %
```

Create a dataframe with the variables with largest memory footpring

```
lst = [*range(10000)]
```

```
utils.mem.top_mem_vars(globals())
```

06 - Exploring Integrations

0. Required library installations

For the execution of one of the integration functions that will be presented here, the geopandas library needs to be installed. To obtain some data for demonstrating the functions, the omnsx library also needs to be installed

```
conda install geopandas osmnx
```

1. Imports

```
import pymove as pm
from pymove.utils import integration as it
from pymove.visualization import folium
import numpy as np
import pandas as pd
import geopandas
```

```
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

2. Load Data

```
move_df = pm.read_csv('geolife_sample.csv', nrows=5000)
move_df.head()
```

```
#Tamanho
move_df.shape[0]
```

```
5000
```

Visualization

```
folium.plot_trajectories(move_df)
```

3. Loading points of interest

```
bbox = move_df.get_bbox()
folium.plot_bbox(bbox, color='blue')
```

```
import osmnx as ox

tags = {'amenity':True}
POIs = ox.geometries_from_bbox(north=bbox[0], south=bbox[2], east=bbox[3],␣
↪west=bbox[1], tags=tags)
```

```
POIs.head()
```

Removing unrated (null) points of interest

```
POIs = POIs.dropna(subset=["amenity"], inplace=False)
```

Adapting to the format needed for integration (With labels 'lat' and 'lon' referring to latitude and longitude, respectively)

```
POIs = POIs[POIs['geometry'].type == 'Point']
POIs['lon'] = POIs['geometry'].x
POIs['lat'] = POIs['geometry'].y
```

Visualization

```
m = folium.plot_trajectories(move_df)
folium.plot_poi(POIs, slice_tags=['amenity'], base_map=m, poi_point='blue')
```

4. Integrating Points of Interest into the DataSet

```
df_4 = move_df.copy()
df_4 = it.join_with_pois(df_4, POIs, label_id='osmid', label_poi_name='name')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=746)))
```

Result

```
df_4.head()
```

Point of interest closest to each point of the trajectory

```
df_4['name_poi'].unique()
```

```
array(['()', '', '', '', '', nan, '', '',
       '', '', '', '2nd Place', '', '', 'HSBC',
       '', '', '', 'Paradiso Coffee', '798 bar',
       'Jazz Cafe', 'Hundred Years Cafe', '', '', '', '',
       'China Construction Bank', '', '', '', '', '',
       '', '101', '', 'Yu Xiao Mian Noodles', '',
       "McDonald's", 'Pizza Hut', 'Starbucks', '', ''],
      dtype=object)
```

5. Integrating Points of Interest into the DataSet (Using join_with_pois_optimizer)

Selecting data

```
POIs_5 = POIs[0:10].copy()
POIs_5['type_poi'] = POIs_5['amenity']
df_5 = move_df.copy()
```

```
POIs_5['type_poi'].unique()
```

```
array(['toilets', 'fast_food', 'massage', 'waste_basket', 'cafe',
       'restaurant', 'bank'], dtype=object)
```

Executing the function

```
df_5 = it.join_with_pois(df_5, POIs_5, label_id='osmid', label_poi_name='name')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=10)))
```

```
df_5.head()
```

6. Integrating Points of Interest into the Category-Based DataSet

```
POIs_5
```

```
df_6 = move_df.copy()
df_6 = it.join_with_pois_by_category(df_6, POIs_5, label_category='amenity', label_id=
↪'name')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5000)))
```

```
df_6.head(10)
```

7. Integrating events (points of interest with timestamp) to the DataSet

It integrates a normal dataframe with Points of interest of events, that is, in addition to the labels referring to latitude and longitude, it also has a label referring to the datetime in which the event occurred. In this example, we will assign random date and time values to some POIs to simulate an operation.

```
indexOfPois = np.arange(0, POIs.shape[0], POIs.shape[0]/20, dtype=np.int64)
POIs_events = POIs.iloc[indexOfPois].copy()
```

```
randomIndexOfMoveDf = np.arange(0, move_df.shape[0], move_df.shape[0]/20, dtype=np.
↪int64)
randomMoveDfSlice = move_df.iloc[randomIndexOfMoveDf].copy()
```

```
POIs_events['datetime'] = randomMoveDfSlice['datetime'].copy()
```

```
df_7 = move_df.copy()
```

```
df_7 = it.join_with_events(
    df_7, POIs_events,
    label_date='datetime', time_window=900,
    label_event_id='osmid', label_event_type='amenity'
)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=20)))
```

```
df_7.head()
```

8. Integration with Point of Interest HOME

The Home type contains, in addition to latitude, longitude and id, the address and city labels.

Creating a home point

```
df_8 = move_df.copy()
home_df = df_8.iloc[300:302].copy()
home_df['formatted_address'] = ['Rua1, n02', 'Rua2, n03']
home_df['city'] = ['ChinaTown', 'ChinaTown']
```

Using the function

```
df_8 = it.join_with_home_by_id(df_8, home_df, label_id='id')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=1)))
```

```
df_8.head()
```

9. Merge of HOME with DataSet already integrated with POIs

Integration

```
df_9 = it.join_with_pois(df_8, POIs, label_id='osmid', label_poi_name='name')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=746)))
```

```
df_9 = it.merge_home_with_poi(df_9)
```

```
df_9.head()
```

11. Union functions

They have the purpose of joining several types of POI that mean the same thing, or similar things, in a single type of POI

Union of Banks

Converts POIs of the types “bank_filials”, “bank_agencies”, “bank_posts”, “bank_PAE” and “bank” to a single type: “banks”

```
df_banks = move_df.copy()

#We create POIs with different type_poi that describe different types of banks to test
indexes_bp = np.linspace(0, df_banks.shape[0], 6)
banks_pois = df_banks[df_banks.index.isin(indexes_bp)].copy()
banks_pois['id'] = [0,1,2,3,4]
banks_pois['type_poi'] = ['bancos_filiais', 'bancos_agencias', 'bancos_postos',
↪'bancos_PAE', 'bank']

banks_pois.head()
```

```
#Join with POIs
df_banks = it.join_with_pois(df_banks, banks_pois, label_id='id', label_poi_name=
↪'type_poi')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5)))
```

```
#Result
df_banks.head(10)
```

```
#Checking the amount of each point assigned to each type of poi
bancos_filiais = df_banks.loc[df_banks['name_poi'] == 'bancos_filiais']
bancos_agencias = df_banks.loc[df_banks['name_poi'] == 'bancos_agencias']
bancos_postos = df_banks.loc[df_banks['name_poi'] == 'bancos_postos']
bancos_PAE = df_banks.loc[df_banks['name_poi'] == 'bancos_PA']
bank = df_banks.loc[df_banks['name_poi'] == 'bank']

print("Number of points close to each bank definition")
print("bancos_filiais: ", bancos_filiais.shape[0])
print("bancos_agencias: ", bancos_agencias.shape[0])
print("bancos_postos: ", bancos_postos.shape[0])
print("bancos_PAE: ", bancos_PAE.shape[0])
print("bank: ", bank.shape[0])
```

```
Number of points close to each bank definition
bancos_filiais:  579
bancos_agencias:  1407
bancos_postos:  916
bancos_PAE:  0
bank:  1238
```

```
#Finally, the Union
df_banks = it.union_poi_bank(df_banks, label_poi="name_poi")

#Result
df_banks.head()
```

```
#Checking
df_banks.loc[df_banks['name_poi'] == 'banks'].shape[0]
```

```
5000
```

Union of Bus Stations

Converts "transit_station" and "bus_points" POIs to a single type: "bus_station"

```
df_bus = move_df.copy()


#We create POIs with different name_poi that describe different types of bus stops to␣
↪test
indexes_bp = np.linspace(0, df_bus.shape[0], 6)
bus_pois = df_bus[df_bus.index.isin(indexes_bp)].copy()
bus_pois['id'] = [0,1,2,3,4]
bus_pois['name_poi'] = ['transit_station', 'transit_station', 'pontos_de_onibus',
↪'transit_station', 'pontos_de_onibus']

#Result
bus_pois.head()
```

```
#Integration
df_bus = it.join_with_pois(df_bus, bus_pois, label_id='id', label_poi_name='name_poi')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=5)))
```

```
#Result
df_bus.head()
```

```
transit_station = df_bus.loc[df_bus['name_poi'] == 'transit_station']
pontos_de_onibus = df_bus.loc[df_bus['name_poi'] == 'pontos_de_onibus']

print("Number of points near transit_station's: ", transit_station.shape[0])
print("Number of points close to pontos_de_onibus's: ", pontos_de_onibus.shape[0])
```

```
Number of points near transit_station's:  2846
Number of points close to pontos_de_onibus's:  2154
```

```
#The union function
df_bus = it.union_poi_bus_station(df_bus, label_poi="name_poi")

df_bus.head()
```

```
#Checking

df_bus.loc[df_bus['name_poi'] == 'bus_station'].shape[0]
```

```
5000
```

Union of Bars and Restaurants

Converts "bar" and "restaurant" POIs to a single type: "bar-restaurant"

```
df_bar = move_df.copy()

#We create POIs with both types
indexes_br = np.linspace(0, df_bar.shape[0], 5)
br_POIs = df_bar[df_bar.index.isin(indexes_br)].copy()
br_POIs['name_poi'] = ['bar','restaurant','restaurant', 'bar']

#Result
br_POIs.head()
```

```
#Integration
df_bar = it.join_with_pois(df_bar, br_POIs, label_id='id', label_poi_name='name_poi')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=4)))
```

```
#Result
df_bar.head()
```

```
#Number of points close to each type
bar = df_bar.loc[df_bar['name_poi'] == 'bar']
restaurant = df_bar.loc[df_bar['name_poi'] == 'restaurant']

print("Closest type points 'bar': ", bar.shape[0])
print("Closest type points 'restaurant': ", restaurant.shape[0])
```

```
Closest type points 'bar':  2539
Closest type points 'restaurant':  2461
```

```
#Union of the two types of POIs into a single
df_bar = it.union_poi_bar_restaurant(df_bar, label_poi="name_poi")

#Result
df_bar.head()
```

```
#Checking
df_bar.loc[df_bar['name_poi'] == 'bar-restaurant'].shape[0]
```

```
5000
```

Union of Parks

Converts “pracas_e_parques” and “park” POIs to a single type: “parks”

```
df_parks = move_df.copy()

#We create POIs with both types
indexes_p = np.linspace(0, df_parks.shape[0], 5)
p_POIs = df_parks[df_parks.index.isin(indexes_p)].copy()
p_POIs['name_poi'] = ['pracas_e_parques','pracas_e_parques','park', 'park']

#Result
p_POIs.head()
```

```
#Integration
df_parks = it.join_with_pois(df_parks, p_POIs, label_id='id', label_poi_name='name_poi
↪')

#Result
df_parks.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=4)))
```

```
#Number of points close to each type of POI
pracas_e_parques = df_parks.loc[df_parks['name_poi'] == 'pracas_e_parques']
park = df_parks.loc[df_parks['name_poi'] == 'park']

print("Number of points closest to pracas_e_parques: ", pracas_e_parques.shape[0])
print("Number of points closest to park: ", park.shape[0])
```

```
Number of points closest to pracas_e_parques:  2716
Number of points closest to park:  2284
```

```
#Union function
df_parks = it.union_poi_parks(df_parks, label_poi="name_poi")

df_parks.head()
```

```
#Checking the new quantity
df_parks.loc[df_parks['name_poi'] == 'parks'].shape[0]
```

```
5000
```

Union of police points

```
df_police = move_df.copy()

#We create POIs with both types
indexes_pol = np.linspace(0, df_police.shape[0], 5)
pol_POIs = df_police[df_police.index.isin(indexes_pol)].copy()
pol_POIs['name_poi'] = ['distritos_policiais','police','distritos_policiais',
↪'distritos_policiais']
```

(continues on next page)

(continued from previous page)

```
#Result
pol_POIs.head()
```

```
#Integration
df_police = it.join_with_pois(df_police, pol_POIs, label_id='id', label_poi_name=
↪'name_poi')

df_police.head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=4)))
```

```
#Quantity of points closest to each type of point
distritos_policiais = df_police.loc[df_police['name_poi'] == 'distritos_policiais']

print("Number of points closest to distritos_policiais: ", distritos_policiais.
↪shape[0])
```

```
Number of points closest to distritos_policiais:  3420
```

```
#Union funcion
df_police = it.union_poi_police(df_police, label_poi="name_poi")
```

```
#Result
df_police.head()
```

```
#Checking
df_police.loc[df_police['name_poi'] == 'police'].shape[0]
```

```
5000
```

12. Integração entre trajetórias e áreas coletivas

```
df_pd = pd.read_csv('geolife_sample.csv')
df_12 = df_pd[0:2000]
gdf = geopandas.GeoDataFrame(df_12, geometry=geopandas.points_from_xy(df_12.lon, df_
↪12.lat))
gdf.head()
```

```
#Creating collective areas
indexes_ac = np.linspace(0, gdf.shape[0], 5)
area_c = df_12[df_12.index.isin(indexes_ac)].copy()
area_c
```

```
#Integration
gdf = it.join_collective_areas(gdf, area_c)
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=4)))
```

```
gdf.head()
```

07 - Exploring Query

1. Imports

```
import pandas as pd
import pymove as pm
from pymove import folium, MoveDataFrame
from pymove.query import query
```

2. Load Data

DataSet - Hurricanes and Typhoons: The NHC publishes the tropical cyclone historical database in a format known as HURDAT, short for HURricane DATabase

```
hurricanes_pandas_df = pd.read_csv('atlantic.csv')
hurricanes_pandas_df
```

```
#Select hurricanes from 2012 to 2015
hurricanes_pandas_df = hurricanes_pandas_df.loc[hurricanes_pandas_df['Date'] >=␣
↪20120000]
hurricanes_pandas_df = hurricanes_pandas_df.loc[hurricanes_pandas_df['Date'] <␣
↪20160000]
hurricanes_pandas_df.shape
```

```
(1639, 22)
```

```
hurricanes_pandas_df[['ID', 'Name', 'Latitude', 'Longitude', 'Date', 'Time']].head()
```

```
hurricanes_pandas_df = pm.conversions.lat_and_lon_decimal_degrees_to_decimal(
    hurricanes_pandas_df, latitude='Latitude', longitude='Longitude'
)

def convert_to_datetime(row):
        this_date = '{}-{}-{}'.format(str(row['Date'])[0:4], str(row['Date'])[4:6],␣
↪str(row['Date'])[6:])
        this_time = '{:02d}:{:02d}:00'.format(int(row['Time']/100), int(str(row['Time
↪'])[-2:]))
        return '{} {}'.format(this_date, this_time)
hurricanes_pandas_df['Datetime'] = hurricanes_pandas_df.apply(convert_to_datetime,␣
↪axis=1)

hurricanes_pandas_df[['ID', 'Name', 'Latitude', 'Longitude', 'Datetime']].head()
```

```
#Converting the pandas dataframe to pymove's MoveDataFrame
hurricanes_2012 = MoveDataFrame(
    data=hurricanes_pandas_df, latitude='Latitude', longitude='Longitude',datetime=
↪'Datetime', traj_id='Name'
)
print(type(hurricanes_2012))
hurricanes_2012.head()
```

```
<class 'pymove.core.pandas.PandasMoveDataFrame'>
```

Visualization

```
folium.plot_trajectories(hurricanes_2012, zoom_start=2)
```

```
#Total hurricane amount between 2012 and 2015
this_ex = hurricanes_2012
this_ex['id'].unique().shape[0]
```

```
55
```

```
#Selecting a hurricane for demonstration
gonzalo = hurricanes_2012.loc[hurricanes_2012['id'].str.strip() == 'GONZALO']
gonzalo.shape
```

```
(39, 23)
```

```
folium.plot_trajectories(
    gonzalo, lat_origin=gonzalo['lat'].median(), lon_origin=gonzalo['lon'].median(),␣
↪zoom_start=2
)
```

2. Range Query

Using distance MEDP (Mean Euclidean Distance Predictive)

```
prox_Gonzalo = query.range_query(gonzalo, hurricanes_2012, min_dist=200, distance=
↪'MEDP')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=55)))
```

```
folium.plot_trajectories(prox_Gonzalo, zoom_start=3)
```

Using Distance MEDT (Mean Euclidean Distance Trajectory)

```
prox_Gonzalo = query.range_query(gonzalo, hurricanes_2012, min_dist=1000, distance=
↪'MEDT')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=55)))
```

```
folium.plot_trajectories(prox_Gonzalo, zoom_start=3)
```

3. KNN (K-Nearest-Neighbor)

Using distance MEDP (Mean Euclidean Distance Predictive)

```
prox_Gonzalo = query.knn_query(gonzalo, hurricanes_2012, id_='id', k=5, distance='MEDP
↪')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=55)))
```

```
folium.plot_trajectories(prox_Gonzalo, zoom_start=3)
```

Using Distance MEDT (Mean Euclidean Distance Trajectory)

```
prox_Gonzalo = query.knn_query(gonzalo, hurricanes_2012, id_='id', k=5, distance='MEDT
↪')
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=55)))
```

```
folium.plot_trajectories(prox_Gonzalo, zoom_start=3)
```

08 - Exploring Semantic

1. Imports

```
import pymove as pm
from pymove.semantic import semantic
```

2. Load Data

```
move_df = pm.read_csv('geolife_sample.csv')
move_df.head()
```

Detect outlier points considering distance traveled in the dataframe

```
outliers = semantic.outliers(move_df)
outliers[outliers['outlier']]
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
move_df.get_bbox()
```

```
(22.147577, 113.548843, 41.132062, 121.156224)
```

Detect points outside of a bounding box

```
fake_bbox = (20, 110, 40, 120)
out_bbox = semantic.create_or_update_out_of_the_bbox(move_df, fake_bbox)
out_bbox[out_bbox['out_bbox']]
```

Detects points with no gps signal, given by the time between adjacent points

```
deactivated = semantic.create_or_update_gps_deactivated_signal(move_df)
deactivated[deactivated['deactivated_signal']]
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

Detects points with jumps, defined by the maximum distance between adjacent points

```
jump = semantic.create_or_update_gps_jump(move_df, )
print(jump[jump['gps_jump']].shape)
jump[jump['gps_jump']].head()
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
(46, 8)
```

Determines if a point belongs to a short trajectory.

```
short = semantic.create_or_update_short_trajectory(move_df)
short[short['short_traj']]
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

```
VBox(children=(HTML(value=''), IntProgress(value=0, max=2)))
```

Python Module Index

p

Index

D

E

F

G

H

I

J

K

L

R

S

X

Y