normal [LE,RO]1 [LO] [RE] [LE,RO],

pyCLARA

Kais Siala
Waleed Sattar Khan
Mohammad Youssef Mahfouz

Version 1.0.0

Jul 09, 2021

Contents

Chapter 1

User manual

1.1 Installation

Note: We assume that you are familiar with git and conda.

First, clone the git repository in a directory of your choice using a Command Prompt window:

```
$ ~\directory-of-my-choice> git clone https://github.com/tum-ens/pyCLARA.git
```

We recommend using conda and installing the environment from the file `geoclustering.yml` that you can find in the repository. In the Command Prompt window, type:

```
$ cd pyCLARA\env\
$ conda env create -f geoclustering.yml
```

Then activate the environment:

```
$ conda activate geoclustering
```

In the folder `code`, you will find multiple files:

File	Description
config.py	used for configuration, see below.
runme.py	main file, which will be run later using `python runme.py`.
lib\initialization.py	used for initialization.
lib\create_subproblems.py	used to split the clustering problem into smaller subproblems.
lib\kmeans_functions.py	includes functions related to the k-means clustering algorithm.
lib\max_p_functions.py	includes functions related to the max-p clustering algorithm.
lib\lines_clustering_functions.py	includes functions for the hierarchical clustering of transmission lines.
lib\spatial_functions.py	contains helping functions for spatial operations.
lib\util.py	contains minor helping functions and the necessary python libraries to be imported.

1.2 config.py

This file contains the user preferences, the links to the input files, and the paths where the outputs should be saved. The paths are initialized in a way that follows a particular folder hierarchy. However, you can change the hierarchy as you wish.

1.2.1 Main configuration function

`config.`**`configuration`**`()`
This function is the main configuration function that calls all the other modules in the code.

Return (paths, param) The dictionary paths containing all the paths to inputs and outputs, and the dictionary param containing all the user preferences.

Return type tuple(dict, dict)

`config.`**`general_settings`**`()`
This function creates and initializes the dictionaries param and paths. It also creates global variables for the root folder `root`, and the system-dependent file separator `fs`.

Return (paths, param) The empty dictionary paths, and the dictionary param including some general information.

Return type tuple(dict, dict)

Note: Both *param* and *paths* will be updated in the code after running the function *config.configuration*.

Note: `root` points to the directory that contains all the inputs and outputs. All the paths will be defined relatively to the root, which is located in a relative position to the current folder.

`config.`**`scope_paths_and_parameters`**`(`*paths*, *param*`)`
This function assigns a name for the geographic scope, and collects information regarding the input rasters that will be clustered:

- *region_name* is the name of the geographic scope, which affects the paths where the results are saved.
- *spatial_scope* is the path to the geographic scope that will be used to clip the map of transmission lines. You can ignore it if you are only clustering rasters.
- *raster_names* are the name tags of the inputs. Preferably, they should not exceed ten (10) characters, as they will be used as attribute names in the created shapefiles. If the user chooses strings longer than that, they will be cut to ten characters, and no error is thrown. The name tags are entered as keys into the dictionary `inputs`.
- *inputs* are the paths to the input rasters (strings). They are given as the first element of a values tuple for each key in the dictionary `inputs`.
- *agg* are the aggregation methods for the input data (strings: either `'mean'` or `'sum'` or `'density'`). They are given as the second element of a values tuple for each key in the dictionary `inputs`.
- *weights* are the weights of the input data during the clustering (int or float). They are given as the third element of a values tuple for each key in the dictionary `inputs`.

Parameters

- **`paths`** (*dict*) – Dictionary including the paths.
- **`param`** (*dict*) – Dictionary including the user preferences.

Return (paths, param) The updated dictionaries paths and param.

Return type tuple(dict, dict)

1.2.2 User preferences

config.**computation_parameters**(*param*)

This function sets the limit to the number of processes *n_jobs* that can be used in k-means clustering.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

config.**kmeans_parameters**(*param*)

This function sets the parameters for the k-means clustering:

- *method*: Currently, two methods for setting the number of clusters are used. By choosing `'maximum_number'`, the user sets the total number of clusters for all parts. This number will be distributed over the parts depending on their size and the standard deviation of their data. If the user chooses `'reference_part'`, then the part with the highest product of relative size and standard deviation (relatively to the maximum) is chosen as a reference part. For this one, the maximum number of clusters is identified using the Elbow method. The optimum number of clusters for all the other parts is a function of that number, and of their relative size and standard deviation.

> **Warning:** The `'maximum_number'` might be exceeded due to the rounding of the share of each part.

- *ratio_size_to_std*: This parameter decides about the weight of the relative size and the relative standard deviation (relatively to the maximum) in determining the optimal number of clusters for each part. A ratio of 7:3 means that 70% of the weight is on the relative size of the part, and 30% is on its standard deviation. Any number greater than zero is accepted.
- *reference_part*: This is a dictionary that contains the parameters for the Elbow method. Cluster sizes between *min* and *max* with a certain *step* will be tested in about for-loop, before the optimal number of clusters for the reference part can be identified. The dictionary is only needed it the method is `'reference_part'`.
- *maximum_number*: This integer sets the maximum number of kmeans clusters for the whole map. It is only used if the method is `'maximum_number'`.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

config.**maxp_parameters**(*param*)

This function sets the parameters for max-p clustering. Currently, one or two iterations of the max-p algorithm can be used, depending on the number of polygons after kmeans.

- *maximum_number*: This number (positive float or integer) defines the maximum number of clusters that the max-p algorithm can cluster in one go. For about 1800 polygons, the calculation takes about 8 hours. The problem has a complexity of $O(n^3)$ in the Bachmann-Landau notation.
- *final_number*: This integer defines the number of clusters that the user wishes to obtain at the end. There is no way to force the algorithm to deliver exactly that number of regions. However, the threshold can be defined as a function of *final_number*, so that the result will be close to it.
- *use_results_of_maxp_parts*: This parameter should be set to zero, unless the user has already obtained results for the first run of the max-p algorithm, and want to skip it and just run the second round. In that case, the user should set the value at 1.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

config.**raster_cutting_parameters**(*paths*, *param*)

This function sets how the large input rasters are cut before starting the clustering. There are two options: the maps are either cut using a shapefile of (multi)polygons, or using rectangular boxes.

- *use_shapefile*: if 1, a shapefile is used, otherwise rectangular boxes.
- *subregions*: the path to the shapefile of (multi)polygons that will cut the large raster in smaller parts (only needed if *use_shapefile* is 1).
- *rows*: number of rows of boxes that the raster will be cut into (only needed if *use_shapefile* is 0).
- *cols*: number of columns of boxes that the raster will be cut into (only needed if *use_shapefile* is 0).

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.

Return (paths, param) The updated dictionaries paths and param.

Return type tuple(dict, dict)

config.**raster_parameters**(*param*)

This function sets the parameters for the input rasters.

- *minimum_valid* is the lowest valid value. Below it, the data is considered NaN.
- *CRS* is the coordinates reference system. It must be the same for all raster input maps.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

config.**transmission_parameters**(*param*)

This function sets the parameters for transmission line clustering.

- *CRS_grid*: The coordinates reference system of the shapefile of transmission lines, in order to read it correctly.
- *default_cap_MVA*: Line capacity in MVA for added lines (to connect electric islands).
- *default_line_type*: Line type for added lines (to connect electric islands).
- *number_clusters*: Target number of regions after clustering, to be used as a condition to stop the algorithm.
- *intermediate_number*: List of numbers of clusters at which an intermediate shapefile will be saved. The values affect the path *grid_intermediate*.
- *debugging_number*: Number of clusters within an intermediate shapefile, that can be used as an input (for debugging). It affects the path *grid_debugging*.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

1.2.3 Paths

config.**output_folders**(*paths*, *param*)

This function defines the paths to multiple output folders:

- *region* is the main output folder. It contains the name of the scope, and the names of the layers used for clustering (as a subfloder).
- *sub_rasters* is a subfolder containing the parts of the input rasters after cutting them.
- *k_means* is a subfolder containing the results of the kmeans clustering (rasters).
- *polygons* is a subfolder containing the polygonized kmeans clusters.
- *parts_max_p* is a subfolder containing the results of the first round of max-p (if there is a second round).
- *final_output* is a subfolder containing the final shapefile.
- *lines_clustering* is a subfolder containing the intermediate and final results of the line clustering.

All the folders are created at the beginning of the calculation, if they do not already exist,

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.

Return paths The updated dictionary paths.

Return type dict

config.**output_paths**(*paths*, *param*)

This function defines the paths of some of the files that will be saved:

- *input_stats* is the path to a CSV file with general information such as the number of parts, the maximal size and the maximal standard deviation in the parts, and the maximum number of clusters as set by the user / by the Elbow method.
- *non_empty_rasters* is the path to a CSV file with information on each subraster (relative size, standard deviation, etc.).
- *kmeans_stats* is the path to a CSV file that is only created if the Elbow method is used (i.e. if using a reference part). It contains statistics for kmeans while applying the Elbow method.
- *polygonized_clusters* is the path to the shapefile with the polygonized rasters for the whole scope.
- *max_p_combined* is the path to the shapefile that is generated after a first round of the max-p algorithm (if there is a second).
- *output* is the path to the shapefile that is generated at the end, i.e. after running max_p_whole_map in `lib.max_p_functions.py`.

For line clustering, the keys start with *grid_*:

- *grid_connected* is the path to the shapefile of lines after adding lines to connect island grids.
- *grid_clipped* is the path to the shapefile of lines after clipping it with the scope.
- *grid_voronoi* is the path to the shapefile of voronoi polygons made from the points at the start/end of the lines.
- *grid_debugging* is the path of an intermediate file during the clustering of regions based on their connectivity.
- *grid_regions* is the path to the final result of the clustering (shapefile of regions based on their connectivity).
- *grid_bottlenecks* is the path to the final result of the clustering (shapefile of transmission line bottlenecks).

Parameters **paths** (*dict*) – Dictionary including the paths.

Return paths The updated dictionary paths.

Return type dict

1.3 runme.py

`runme.py` calls the main functions of the code:

```
from lib.initialization import initialization
from lib.create_subproblems import cut_raster
from lib.kmeans_functions import *
from lib.max_p_functions import *
from lib.lines_clustering_functions import *
from lib.util import *

if __name__ == "__main__":

    paths, param = initialization()

    cut_raster(paths, param)

    # k-means functions
    calculate_stats_for_non_empty_rasters(paths, param)
    if param["kmeans"]["method"] == "reference_part":
        choose_ref_part(paths)
        identify_max_number_of_clusters_in_ref_part(paths, param)
    k_means_clustering(paths, param)
    polygonize_after_k_means(paths, param)

    # max-p functions
    max_p_clustering(paths, param)

    # lines clustering functions
    lines_clustering(paths, param)
```

1.4 Recommended input sources

For a list of GIS data sources, check this wikipedia article.

1.4.1 Shapefile of transmission lines

High-voltage power grid data for Europe and North America can be obtained from GridKit, which used OpenStreetMap as a primary data source.

In this repository, the minimum requirements are a shapefile of lines with the attributes ID, Cap_MVA and Type. Such a file can be obtained using the repository tum-ens/pyPRIMA to clean the GridKit data.

1.4.2 Shapefile of the region of interest (useful for lines clustering)

The shapefile allows to identify the lines that are inside the scope, and those that lie outside of it. Hence, only the outer borders of the polygons matter. If you are interested in administrative divisions, you may consider downloading the shapefiles from the website of the Global Administration Divisions (GADM). You can also create your own shapefiles using a GIS software.

1.4.3 High resolution rasters

Any raster can be used. If multiple rasters are used, ensure that they have the same resolution, the same projection, and the same geographic extent. Rasters for renewable potentials can be generated using the GitHub repository tum-ens/pyGRETA. A high resolution raster (30 arcsec) of population density can be downloaded from the website of SEDAC after registration. A high resolution raster (15 arcsec = 1/240° longitude and 1/240° latitude) made of 24 tiles can be downloaded from viewfinder panoramas.

1.5 Recommended workflow

The script is designed to be modular and split into four multiple modules: `lib.create_subproblems`, `lib.kmeans_functions`, `lib.max_p_functions`, and `lib.lines_clustering_functions`.

> **Warning:** The outputs of some modules serve as inputs to the others (this applies to the first three modules). Therefore, the user will have to run the script sequentially.

There are two recommended use cases: either the clustering of rasters, or the clustering of transmission lines. The modules will be presented in the order in which the user will have to run them.

1. *Clustering of rasters*

a. *Creating subproblems*

b. *k-means clustering*

c. *max-p functions*

2. *Clustering of transmission lines*

It is recommended to thoroughly read through the configuration file *config.py* and modify the input paths and computation parameters before starting the *runme.py* script. Once the configuration file is set, open the *runme.py* file to define what use case you will be using the script for.

1.5.1 Clustering of rasters

The first use case deals with the clustering of high resolution rasters. Depending on the size of the rasters and the user preferences, all or some of these modules will be used:

Creating subproblems

Instead of applying the clustering algorithms directly on the original rasters, the module *lib.create_subproblems* is called to split the rasters into smaller ones. There are two options: either to split the rasters into rectangles of similar sizes, or according to polygon shapes provided by the user.

Note: If you would like to cluster data on a European level, but would like to do it for each country separately, provide a shapefile of European countries to define the subregions.

You can also skip this step altogether and cluster the whole dataset at once (not recommended for very large maps, because the quality of the results and the speed of the calculation are affected).

k-means clustering

This step is also optional. The purpose is to compress the amount of information so that the max-p algorithm can be applied. The max-p algorithm can cluster up to ~1800 polygons in about 8h. So the functions inside the *lib.kmeans_functions* module have two goals: reduce the number of data points while preserving the maximum amount of information that can be processed by the next module, and polygonizing the rasters (i.e. create shapefiles of polygons from clustered rasters).

The user can define the number of clusters k, or let the code decide depending on the standard deviation and the number of valid data points.

max-p functions

The max-p algorithm is the one that ensures the spatial contiguity of the clustered data points, but due to its computational complexity, the previous steps might be needed. Also, there is a possibility to run the max-p algorithm in two steps: first on each subproblem separately, then on the whole geographic scope after combining the results of the subproblems. If the number of polygons after k-means is manageable in one run, then the max-p algorithm is applied once. The result is a shapefile of contiguous polygons with similar characteristics.

Note: You can customize the properties, the weights of the data sets, and the aggregating functions to be used during the clustering, e.g. 20% solar FLH (averaged), 40% wind FLH (averaged), and 40% electricity demand (summed).

1.5.2 Clustering of transmission lines

This use case is currently independent of the previous one. Starting from the maximum number of regions surrounding each node in the grid, it uses a hierarchical algorithm to merge connected nodes that have a high transmission capacity flowing from/into them and a low area. The algorithm stops when the target number of clusters is reached.

Chapter 2

Theory

2.1 Introduction

A number of different algorithms have been developed by scientists for clustering geographic data. The most trivial and commonly used algorithm is the k-means algorithm but it has some limitations. These limitations urged the researchers to come up with more novel and sophisticated methods of clustering geographic data. Usually, the type of algorithm which is used depends on the type of data available and the associated time complexity and requirements for clustering quality. Furthermore, the clustering algorithms are based on different techniques and one of these techniques is hierarchical clustering.

2.2 Hierarchical Clustering

A hierarchical clustering algorithm forms a dendrogram which is basically a tree that splits the input data into small subsets using a recursive approach [1]. The dendrogram can be built using two different approaches which are “bottom-up” or “top-down” approach. As the name suggests, the bottom-up approach treats each object as a separate cluster and combines the two “closest” clusters into one. The algorithm continues until all objects are merged together to form one big object. On the other hand, the top-down approach starts with one big cluster in which all the objects exist. The big cluster is repeatedly broken down into smaller clusters in each iteration of the algorithm [1].

2.3 k-means Method

The k-means method is one of the oldest methods that has been used for clustering. In this method, the average value of data objects in a cluster is used as the cluster center [2]. The steps that the k-means method follows are:

The number of clusters k are chosen. The choice of right k is important as small k can result in clusters in which distances to centroid are very long whereas a too large k means that the algorithm will take longer to converge with little to no improvement in average distances. # Each cluster is initialized by either of the two methods namely Forgy method or Random partition method. For standard k-means algorithm, Forgy method is preferable in which k random observations are chosen as initial means for the clusters [3]. # This step is known as the assignment step. A total of k number of clusters are created with each data object being assigned to the nearest mean. The figure so obtained is known as Voronoi diagram. # The mean of all the data points in each cluster is calculated and the newly calculated mean serves as the new centroid for each cluster. This step is called the update step. # Step (3) and (4) are repeated until convergence criteria has been attained. Usually, the convergence is achieved when the object assignments do not change any more.

Generally, the k-means algorithm has a time complexity equal to O (n2) [4]. Moreover, it is a heuristic method which means that convergence to a global optimum is not guaranteed. However, as k-means is a relatively fast-algorithm, it is a common practice to run k-means method of clustering with different starting conditions and choose the run that has the best results.

2.4 k-means++ Method

The k-means++ clustering algorithm can be thought of as an add-on to the standard k-means clustering algorithm. The k-means++ algorithm is designed to improve the quality of clusters that are formed as local optimum using the standard k-means method [4]. Moreover, this method also tends to lower the runtime of the subsequently used k-means algorithm by making it converge quickly. k-means++ achieves this by choosing "seeds" for k-means algorithm. The "seeds" are the initial k cluster centers which are chosen based on a certain criteria instead of randomly choosing the cluster centers as was the case in the standard k-means algorithm. The steps that k-means++ algorithm follows for initializing the cluster centers are as follows:

Choose one data point at random from the given data set and consider it as the first center c¬i. # Calculate the distance of each of the data points in the set from the nearest cluster center with D(x) being the distance of each point. # Choose a new cluster center ci. # Repeat steps (2) and (3) until a k number of centers have been chosen. # Using the initialized centers, proceed with the standard k-means algorithm as outlined in previous section.

Although initializing cluster centers using k-means++ method is computationally expensive, the subsequent k-means method actually takes less time to converge and as a result, the overall time taken to converge is usually less as compared to the standard k-means method. Moreover, the k-means++ method is O (log k) competitive [4]. Furthermore, as mentioned above, the local optimum obtained is much more optimized and therefore, the k-means++ method is better than the standard k-means method for clustering of data. Fig. 3 [5] shows the results of a clustering performed on a set of typical data using k-means and k-means++ method. It is clear from the figure that error is smaller when k-means++ is used for clustering and the convergence time is also almost the same.

2.5 max-p Method

The max-p regions problem has been developed to satisfy spatial contiguity constraints while clustering. As opposed to other constraint-based methods, this methods models the total number of sectors as an internal parameter [6]. Moreover, this approach does not put restrictions on the geometry of regions. In fact, it lets the data sets dictate the shape. The heuristic solution that has been proposed in order to solve the max-p regions problem follows the following steps.

The first is the construction phase which is subdivided into two more phases. The first sub-phase is named growing phase in which the algorithm selects an unassigned area at random. This is called the seed area. # The neighboring regions of seed area are continuously added to the first seed until a threshold is reached. # Then the algorithm chooses a new seed area and grows a region by following step (2). This step is repeated until no new seeds which can satisfy the threshold value can be found. # Areas which are not assigned to a region are called enclaves. Therefore, at the end of growing phase, a set of enclaves and growing regions are formed. # The number of growing regions and enclaves can change in successive iterations. Solutions in which maximum growing regions are equal to maximum growing regions in the previous iteration are only kept. # The next step is the process of enclave assignment and each solution found in previous steps is passed to this step. Each enclave is assigned to a neighboring region based on similarity. This marks the end of construction phase. # In the next step i.e. the local search phase, a set of neighboring solutions is obtained by modifying the found solutions and improving them. Neighboring solutions are created in a way that each solution is feasible. One way to create neighboring solutions is to move one region to another region [7]. # Finally, a local search algorithm such as Simulated Annealing, Greedy Algorithm or Tabu Search Algorithm is used to improve the solution.

The main purpose of max-p regions problem and the proposed solution is to aggregate small areas into homogeneous regions in a way that the value of a spatially extensive attribute is always more than the threshold value. It can be useful for clustering rasters in which constraints such as population density, energy per capita need to be met [6].

Chapter 3

Implementation

Start with the configuration:

You can run the code by typing:

```
$ python runme.py
```

The script `runme.py` calls the main functions of the code, which are explained in the following sections.

3.1 initialization.py

`lib.initialization.`**`initialization`**`()`

This function reads the user-defined parameters and paths from `config.py`, then checks the validity of the input files. If they are missing or are not rasters, a warning is thrown and the code is exited. The same applies if the rasters do not have the same resolution or scope.

If the input files are valid, the CSV file in *input_stats* is generated and filled with the information that is already available. It will be called by other functions and eventually filled by them.

Returns The updated dictionaries param and paths.

Return type tuple(dict, dict)

3.2 create_subproblems.py

`lib.create_subproblems.`**`cut_raster`**`(`*paths*, *param*`)`

This function decides, based on the user preference in *use_shapefile*, whether to call the module `cut_raster_using_shapefile` or `cut_raster_using_boxes`.

Parameters

- **`paths`** (*`dict`*) – Dictionary of paths to inputs and outputs.
- **`param`** (*`dict`*) – Dictionary of parameters and user preferences.

Returns The submodules `cut_raster_using_shapefile` and `cut_raster_using_boxes` have their own outputs.

Return type None

`lib.create_subproblems.`**`cut_raster_using_boxes`**`(`*paths*, *param*`)`

This function cuts the raster file into a m*n boxes with m rows and n columns.

Parameters

- **paths** (*dict*) – The paths to the input rasters *inputs*, to the output raster parts *sub_rasters*, and to the CSV file *input_stats* which will be updated.
- **param** (*dict*) – Parameters that include the number of *rows* and *cols*, and the *raster_names*.

Returns The raster parts are saved as GEOTIFF files, and the CSV file *input_stats* is updated.

Return type None

lib.create_subproblems.**cut_raster_using_shapefile**(*paths*, *param*)
This function cuts the raster file into parts using the features of a shapefile of (multi)polygons. Each feature is used as a mask, so that only the pixels lying on it are extracted and saved in a separate raster file.

Parameters

- **paths** (*dict*) – The paths to the input rasters *inputs*, to the shapefile of *subregions*, to the output raster parts *sub_rasters*, and to the CSV file *input_stats* which will be updated.
- **param** (*dict*) – Parameters that include the array *Crd_all*, the array *res_desired* and the dictionary *GeoRef* for georeferencing the output rasters, the minimum valid value *minimum_valid*, and the *raster_names*.

Returns The raster parts are saved as GEOTIFF files, and the CSV file *input_stats* is updated.

Return type None

3.3 kmeans_functions.py

class lib.kmeans_functions.**OptimumPoint**(*init_x*, *init_y*)
This class is used in the Elbow method to identify the maximum distance between the end point and the start point of the curve of inertia as a function of number of clusters.

lib.kmeans_functions.**calculate_stats_for_non_empty_rasters**(*paths*, *param*)
This function calculates statistics for all non empty subrasters. These statistics include the number of rows and columns, the size (number of valid points), the standard deviation, the relative size (to the maximum) and the relative standard deviation, the product of the latter two, and the values of four mapping functions using the relative size and relative standard deviation.

The product of the relative size and relative standard deviation is used to identify the reference part. As of the four mapping functions, they are used to identify the four parts that lie in the corners of the cloud of points, where each point represents a part and is plotted on a graph with the relative size in one axis, and relative standard deviation on the other.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *inputs*, to the CSV *input_stats*, to the folder of *sub_rasters*, and to the output CSV *non_empty_rasters*.
- **param** (*dict*) – Dictionary of parameters containing the *raster_names* and the minimum valid value in the rasters, *minimum_valid*.

Returns The results are directly saved in the desired CSV file *non_empty_rasters*, and the CSV file *input_stats* is also updated.

Return type None

lib.kmeans_functions.**choose_ref_part**(*paths*)
This function chooses the reference part for the function identify_max_number_of_clusters_in_ref_part. The reference part is chosen based on the product of relative size and relative standard deviation. The part with the largest product in all the input files is chosen.

Parameters **paths** (*dict*) – The paths to the CSV files *non_empty_rasters* and *input_stats*.

Returns The CSV file *input_stats* is updated.

Return type None

lib.kmeans_functions.**identify_max_number_of_clusters_in_ref_part**(*paths*, *param*)

This function identifies the maximum number of clusters for the reference part using the Elbow method. The number of clusters is varied between *min* and *max* by *step*, and in each case, the inertia (distances to the cluster centers) are calculated. If the slope of the change of the inertia goes below a certain threshold, the function is interrupted and the maximum number of clusters for the reference part is determined.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *inputs*, to the CSV *input_stats*, to the folder of *sub_rasters*, and to the output CSV *kmeans_stats*.
- **param** (*dict*) – Dictionary of parameters containing the *raster_names* and their *weights*, the minimum valid value in the rasters, *minimum_valid*, kmeans-related parameters for the iteration of the Elbow method, and the number of processes *n_job*.

Returns The results are directly saved in the desired CSV file *kmeans_stats*, and the CSV file *input_stats* is also updated.

Return type None

lib.kmeans_functions.**identify_opt_number_of_clusters**(*paths*, *param*, *part*, *size_of_raster*, *std_of_raster*)

This function identifies the optimal number of clusters which will be chosen for k-means in each part.

In case you are using a reference part, then the optimal number is a function of the number of clusters in the reference part, and of the relative size and relative standard deviation, which are weighted according to *ratio_size_to_std*.

In case you are using the maximum number for the whole map, then the optimal number in each part is a function of the total number, of the relative size and relative standard deviation, and the weights in *ratio_size_to_std*.

Parameters

- **paths** (*dict*) – Dictionary of paths pointing to the location of the input CSV file *non_empty_rasters* and to *input_stats*.
- **param** (*dict*) – Dictionary of parameters including the ratio between the relative size and the relative standard deviation *ratio_size_to_std* and the *method* for setting the number of clusters.
- **part** (*integer*) – Counter for the raster parts.
- **size_of_raster** (*integer*) – Number of valid data points in the raster part.
- **std_of_raster** (*float*) – Standard deviation of the data in the raster part.

Return optimum_no_of_clusters_for_raster Optimum number of clusters for the raster part according to the chosen method.

Return type integer

lib.kmeans_functions.**k_means_clustering**(*paths*, *param*)

This function does the k-means clustering for every part.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *inputs*, to the CSV *input_stats* and *non_empty_rasters*, to the folder of *sub_rasters*, and to the output folder *kmeans*.
- **param** (*dict*) – Dictionary of parameters containing the *raster_names* and their *weights* and aggregation methods *agg*, the minimum valid value in the rasters, *minimum_valid*, the *method* for finding the number of kmeans clusters, and the number of processes *n_job*.

Returns The results are directly saved in the desired CSV file *kmeans_stats*, and the CSV file *input_stats* is also updated.

Return type None

lib.kmeans_functions.**polygonize_after_k_means**(*paths*, *param*)
This function converts the rasters created after k-means clustering into shapefiles of (multi)polygons which are used in the max-p algorithm.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *kmeans* for retrieving inputs, to the CSV *non_empty_rasters*, and to the folder *polygons* for saving outputs.
- **param** (*dict*) – Dictionary of parameters containing the minimum valid value in the rasters, *minimum_valid*, and the *CRS* of the shapefiles.

Returns The results are directly saved in the desired paths for each part (folder *polygons*) and for the whole map (file *polygonized_clusters*).

Return type None

3.4 max_p_functions.py

lib.max_p_functions.**correct_neighbors_in_shapefile**(*param*, *combined_file*, *existing_neighbors*)
This function finds the neighbors in the shapefile. Somehow, max-p cannot figure out the correct neighbors and some clusters are physically neighbors but they are not considered as neighbors. This is where this function comes in.

It creates a small buffer around each polygon. If the enlarged polygons intersect, and the area of the intersection exceeds a threshold, then the polygons are considered neighbors, and the dictionary of neighbors is updated.

Parameters

- **param** (*dict*) – The dictionary of parameters including the coordinate reference system *CRS* and the resolution of input rasters *res_desired*.
- **combined_file** (*str*) – The path to the shapefile of polygons to be clustered.
- **existing_neighbors** (*dict*) – The dictionary of neighbors as extracted from the shapefile, before any eventual correction.

Return neighbors_corrected The dictionary of neighbors after correction (equivalent to an adjacency matrix).

Return type dict

lib.max_p_functions.**eq_solver**(*coef*, *ll_point*, *ul_point*, *ur_point*)
This function serves as the solver to find coefficient values A, B and C for our defined function which is used to calculate the threshold.

Parameters

- **coef** (*dict*) – The coefficients which are calculated.
- **ll_point** (*tuple(int, int)*) – Coordinates of lower left point.
- **ul_point** (*tuple(int, int)*) – Coordinates of upper left point.
- **ur_point** (*tuple(int, int)*) – Coordinates of upper right point.

Return f Coefficient values for A, B and C in a numpy array. A is f[0], B is f[1] and C is f[2].

Return type numpy array

lib.max_p_functions.**get_coefficients**(*paths*)
This function gets the coefficients A, B and C for solving the 3 equations which will lead to the calculation of the threshold in the max-p algorithm.

Parameters **paths** (*str*) – The dictionary of paths including the one to *non_empty_rasters*.

Return coef The coefficient values for A, B and C returned as a dictionary. The expected structure is similar to this dictionary: {'A': 0.55, 'B': 2.91, 'C': 0.61}.

Return type dict

lib.max_p_functions.**max_p_clustering**(*paths*, *param*)
This function applies the max-p algorithm to the obtained polygons. Depending on the number of clusters in the whole map after k-means, it decides whether to run max-p clustering multiple times (for each part, then for the whole map) or once (for the whole map). If you have already results for each part, you can skip that by setting *use_results_of_max_parts* to 1.

Parameters

- **paths** (*dict*) – Dictionary of paths pointing to *polygonized_clusters* and *max_p_combined*.
- **param** (*dict*) – Dictionary of parameters including max-p related parameters (*maximum_number* and *use_results_of_maxp_parts*), and eventually the *compression_ratio* for the first round of max-p clustering.

Returns The called functions `max_p_parts` and `max_p_whole_map` generate outputs.

Return type None

lib.max_p_functions.**max_p_parts**(*paths*, *param*)
This function applies the max-p algorithm on each part. It identifies the neighbors from the shapefile of polygons for that part. If there are disconnected parts, it assumes that they are neighbors with the closest polygon.

The max-p algorithm aggregates polygons to a maximum number of regions that fulfil a certain condition. That condition is formulated as a minimum share of the sum of data values, `thr`. The threshold is set differently for each part, so that the largest and most diverse part keeps a large number of polygons, and the smallest and least diverse is aggregated into one region. This is determined by the function `get_coefficients`.

After assigning the clusters to the polygons, they are dissolved according to them, and the values of each property are aggregated according to the aggregation functions of the inputs, saved in *agg*.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *inputs* and *polygons*, to the CSV *non_empty_rasters*, to the output folder *parts_max_p* and to the output file *max_p_combined*.
- **param** (*dict*) – Dictionary of parameters containing the *raster_names* and their *weights* and aggregation methods *agg*, the *compression_ratio* of the polygonized kmeans clusters, and the *CRS* to be used for the shapefiles.

Returns The results of the clustering are shapefiles for each part, saved in the folder *parts_max_p*, and a combination of these for the whole map *max_p_combined*.

Return type None

lib.max_p_functions.**max_p_whole_map**(*paths*, *param*, *combined_file*)
This function runs the max-p algorithm for the whole map, either on the results obtained from `max_p_parts`, or on those obtained from `polygonize_after_k_means`, depending on the number of polygons after kmeans clustering.

It identifies the neighbors from the shapefile of polygons. If there are disconnected components (an island of polygons), it assumes that they are neighbors with the closest polygon. It also verifies that the code identifies neighbors properly and corrects that eventually using `correct_neighbors_in_shapefile`.

The max-p algorithm aggregates polygons to a maximum number of regions that fulfil a certain condition. That condition is formulated as a minimum share of the sum of data values, `thr`. The threshold for the whole map is set as a function of the number of polygons before clustering, and the desired number of polygons at the end. However, that number may not be matched exactly. The user may wish to adjust the threshold manually until the desired number is reached (increase the threshold to reduce the number of regions, and vice versa).

After assigning the clusters to the polygons, they are dissolved according to them, and the values of each property are aggregated according to the aggregation functions of the inputs, saved in *agg*.

Parameters

- **paths** (*dict*) – Dictionary containing the paths to the folder of *inputs* and to the output file *output*.
- **param** (*dict*) – Dictionary of parameters containing the *raster_names* and their *weights* and aggregation methods *agg*, the desired number of features at the end *final_number*, and the *CRS* to be used for the shapefiles.
- **combined_file** (*str*) – Path to the shapefile to use as input. It is either the result obtained from `max_p_parts`, or the one obtained from `polygonize_after_k_means`.

Returns The result of the clustering is one shapefile for the whole map saved directly in *output*.

Return type None

For the hierarchical clustering of the transmission network, use the following script:

3.5 lines_clustering_functions.py

`lib.lines_clustering_functions.`**`clip_transmission_shapefile`**(*paths*, *param*)

This function clips the shapefile of the transmission lines using the shapefile of the scope. MULTILINESTRING instances are formed as a result of clipping. Hence, the script cleans the clipped transmission file by replacing the MULTILINESTRING instances with LINESTRING instances.

Parameters

- **paths** (*dict*) – Dictionary of paths including the path to the shapefile of the transmission network after connecting islands, *grid_connected*, and to the output *grid_clipped*.
- **param** – Dictionary of parameters including *CRS_grid*.

Returns The shapefile of clipped transmission lines is saved directly in the desired path *grid_clipped*.

Return type None

`lib.lines_clustering_functions.`**`cluster_transmission_shapefile`**(*paths*, *param*)

This function clusters the transmission network into a specified number of clusters. It first reads the shapefile of voronoi polygons, and initializes its attributes *elec_neighbors*, *trans_lines*, *Area*, *Cap*, and *Ratio*. Starting with the polygon with the highest ratio, it merges it with its electric neighbors with the highest ratio as well. It then updates the values and repeats the algorithm, until the target number of clusters is reached.

Parameters

- **paths** (*dict*) – Dictionary of paths pointing to *grid_clipped*, *grid_debugging*, *grid_voronoi*, and to the outputs *grid_intermediate* and *grid_regions*.
- **param** (*dict*) – Dictionary containing the parameter *CRS_grid* and the user preferences *number_clusters* and *intermediate_number*.

Returns The intermediate and final outputs are saved directly in the desired shapefiles.

Return type None

lib.lines_clustering_functions.**connect_islands**(*paths*, *param*)
This script takes a shapefile of a transmission network and uses graph theory to identify its components (electric islands). After the identification of islands, it creates connections between them using additional transmission lines with low capacities. This correction assumes that there are actually no electric islands, and that multiple graph components only exist because some transmission lines are missing in the data. The output is a shapefile of transmission lines with no electric islands.

Parameters

- **paths** (*dict*) – Dictionary of paths including the path to the input shapefile of the transmission network, *grid_input*.
- **param** – Dictionary of parameters including *CRS_grid*, *default_cap_MVA*, and *default_line_type*.

Returns The shapefile with connections between islands is saved directly in the desired path *grid_connected*.

Return type None

lib.lines_clustering_functions.**create_voronoi_polygons**(*paths*, *param*)
This function creates a shapefile of voronoi polygons based on the points at the start/end of the lines.

Parameters

- **paths** (*dict*) – Dictionary of paths including the path to the shapefile of the transmission network after clipping, *grid_clipped*, to the scope *spatial_scope*, and to the output *grid_voronoi*.
- **param** – Dictionary of parameters including *CRS_grid*.

Returns The shapefile of voronoi polygons is saved directly in the desired path *grid_voronoi*.

Return type None

lib.lines_clustering_functions.**lines_clustering**(*paths*, *param*)
This function applies the hierarchical clustering algorithm to the shapefile of transmission lines. It first ensures that the whole grid is one component (no electric islands), by eventually adding fake lines with low capacity. Then it clips the grid to the scope, and creates a shapefile of voronoi polygons based on the points at the start/end of the lines. Regions with a small area and high connectivity to their neighbors are aggregated together, until the target number of regions is reached.

Parameters

- **paths** (*dict*) – Dictionary of paths pointing to input files and output locations.
- **param** (*dict*) – Dictionary of parameters including transmission-line-related parameters.

Returns The called functions `connect_islands`, `clip_transmission_shapefile`, `create_voronoi_polygons`, and `cluster_transmission_shapefile` generate outputs.

Return type None

lib.lines_clustering_functions.**update_values_in_geodataframes**(*gdf_trans*, *gdf_voronoi*, *poly1*, *poly2*, *cluster_no*)
This function updates the values in the geodataframes *gdf_trans* and *gdf_voronoi* after dissolving the polygons and is called in the loops in `cluster_transmission_shapefile`.

Parameters

- **gdf_trans** (*geopandas GeoDataFrame*) – Geodataframe of transmission lines, containing the columns *Start_poly* and *End_poly*.
- **gdf_voronoi** (*geopandas GeoDataFrame*) – Geodataframe of polygons, containing the columns *trans_lines*, *elec_neighbors*, *Cluster*, *Cap*, *Area*, and *Ratio*.

- **poly1** (*geopandas GeoDataFrame*) – First polygon to be dissolved, containing the same columns as gdf_voronoi.
- **poly2** (*geopandas GeoDataFrame*) – Second polygon to be dissolved, containing the same columns as gdf_voronoi.
- **cluster_no** (*integer*) – Cluster number to be used for the dissolved polygons.

Return (gdf_trans, gdf_voronoi) Updated geodataframes after dissolving poly1 and poly2.

Return type tuple of geodataframes

Helping functions for the models are included in `spatial_functions.py`, and `util.py`.

3.6 spatial_functions.py

`lib.spatial_functions.`**array2raster**(*newRasterfn*, *array*, *rasterOrigin*, *param*)
This function saves array to geotiff raster format (used in cutting with shapefiles).

Parameters

- **newRasterfn** (*string*) – Output path of the raster.
- **array** (*numpy array*) – Array to be converted into a raster.
- **rasterOrigin** (*list of two floats*) – Latitude and longitude of the Northwestern corner of the raster.
- **param** (*dict*) – Dictionary of parameters including *GeoRef* and *CRS*.

Returns The raster file will be saved in the desired path *newRasterfn*.

Return type None

`lib.spatial_functions.`**array_to_raster**(*array*, *destination_file*, *input_raster_file*)
This function changes from array back to raster (used after kmeans algorithm).

Parameters

- **array** (*numpy array*) – The array which needs to be converted into a raster.
- **destination_file** (*string*) – The file name where the created raster file is saved.
- **input_raster_file** (*string*) – The original input raster file from which the original coordinates are taken to convert the array back to raster.

Returns The raster file will be saved in the desired path *destination_file*.

Return type None

`lib.spatial_functions.`**assign_disconnected_components_to_nearest_neighbor**(*data*, *w*)
This loop is used to force any disconnected group of polygons (graph component) to be assigned to the nearest neighbors.

Parameters

- **data** (*geodataframe*) – The geodataframe of polygons to be clustered.
- **w** (*pysal weights object*) – The pysal weights object of the graph (`w.neighbors` is similar to an adjacency matrix).

Return w The updated pysal weights objected is returned.

Return type pysal weights object

`lib.spatial_functions.`**calc_geotiff**(*Crd_all*, *res_desired*)
This function returns a dictionary containing the georeferencing parameters for geotiff creation, based on the desired extent and resolution.

Parameters

- **Crd_all** (*numpy array*) – Coordinates of the bounding box of the spatial scope.
- **res_desired** (*list*) – Desired data resolution in the vertical and horizontal dimensions.

Return GeoRef Georeference dictionary containing *RasterOrigin*, *RasterOrigin_alt*, *pixelWidth*, and *pixelHeight*.

Return type dict

lib.spatial_functions.**calc_region**(*region*, *Crd_reg*, *res_desired*, *GeoRef*)
This function reads the region geometry, and returns a masking raster equal to 1 for pixels within and 0 outside of the region.

Parameters

- **region** (*Geopandas series*) – Region geometry
- **Crd_reg** (*list*) – Coordinates of the region
- **res_desired** (*list*) – Desired high resolution of the output raster
- **GeoRef** (*dict*) – Georeference dictionary containing *RasterOrigin*, *RasterOrigin_alt*, *pixelWidth*, and *pixelHeight*.

Return A_region Masking raster of the region.

Return type numpy array

lib.spatial_functions.**ckd_nearest**(*gdf_a*, *gdf_b*, *bcol*)
This function finds the distance and the nearest points in gdf_b for every point in gdf_a.

Parameters

- **gdf_a** (*geodataframe*) – GeoDataFrame of points, forming a component that is disconnected from *gdf_b*.
- **gdf_b** (*geodataframe*) – GeoDataFrame of points, forming a component that is disconnected from *gdf_a*.
- **bcol** (*string*) – Name of column that should be listed in the resulting DataFrame.

Return df Dataframe with the combinations of pair of points as rows, and `'distance'` and `'bcol'` as columns.

Return type pandas dataframe

lib.spatial_functions.**crd_bounding_box**(*Crd_regions*, *resolution*)
This function calculates coordinates of the bounding box covering data in a given resolution.

Parameters

- **Crd_regions** (*numpy array*) – Coordinates of the bounding boxes of the regions.
- **resolution** (*numpy array*) – Data resolution.

Return Crd Coordinates of the bounding box covering the data for each region.

Return type numpy array

lib.spatial_functions.**ind_from_crd**(*Crd*, *Crd_all*, *res*)
This function converts longitude and latitude coordinates into indices within the spatial scope of the data.

Parameters

- **Crd** (*numpy array*) – Coordinates to be converted into indices.
- **Crd_all** (*numpy array*) – Coordinates of the bounding box of the spatial scope.
- **res** (*list*) – Resolution of the data, for which the indices are produced.

Return Ind Indices within the spatial scope of data.

Return type numpy array

lib.spatial_functions.**polygonize_raster**(*input_file*, *output_shapefile*, *column_name*)
This function is used to change from a raster to polygons as max-p algorithm only works with polygons.

Parameters

- **input_file** (*string*) – The path to the file which needs to be converted to a polygon from a raster.
- **output_shapefile** (*string*) – The path to the shapefile which is generated after polygonization.
- **column_name** (*string*) – The column name, the values from which are used for conversion.

Returns The shapefile of (multi)polygons is saved directly in the desired path *output_shapefile*.

Return type None

3.7 util.py

lib.util.**create_json**(*filepath*, *param*, *param_keys*, *paths*, *paths_keys*)
Creates a metadata JSON file containing information about the file in filepath by storing the relevant keys from both the param and path dictionaries.

Parameters

- **filepath** (*string*) – Path to the file for which the JSON file will be created.
- **param** (*dict*) – Dictionary of dictionaries containing the user input parameters and intermediate outputs.
- **param_keys** (*list of strings*) – Keys of the parameters to be extracted from the *param* dictionary and saved into the JSON file.
- **paths** (*dict*) – Dictionary of dictionaries containing the paths for all files.
- **paths_keys** (*list of strings*) – Keys of the paths to be extracted from the *paths* dictionary and saved into the JSON file.

Returns The JSON file will be saved in the desired path *filepath*.

Return type None

lib.util.**display_progress**(*message*, *progress_stat*)
This function displays a progress bar for long computations. To be used as part of a loop or with multiprocessing.

Parameters

- **message** (*string*) – Message to be displayed with the progress bar.
- **progress_stat** (*tuple(int, int)*) – Tuple containing the total length of the calculation and the current status or progress.

Returns The status bar is printed.

Return type None

lib.util.**get_x_y_values**(*paths*)
This function finds the rel_size and rel_std of the four corners of the x,y scatter plot between rel_size and rel_std.

Parameters **paths** (*dict*) – Dictionary of paths including the path to the CSV file *non_empty_rasters*.

Returns Coordinates of the upper left, upper right, lower left and lower right points of the x,y scatter plot between rel_size and rel_std.

Return type tuple(tuples(int, int))

lib.util.**timecheck**(**args*)

This function prints information about the progress of the script by displaying the function currently running, and optionally an input message, with a corresponding timestamp. If more than one argument is passed to the function, it will raise an exception.

Parameters **args** (*string*) – Message to be displayed with the function name and the timestamp (optional).

Returns The time stamp is printed.

Return type None

Raise Too many arguments have been passed to the function, the maximum is only one string.

Python Module Index

c

l

Index

O

P

T

U