GeoGraph - Package Documentation

Aug 16, 2022

CONTENTS:

Welcome to the GeoGraph documentation!

CHAPTER

ONE

WHAT IS GEOGRAPH?

The Python package GeoGraph is built around the idea of geospatially referenced graph - a *GeoGraph*. Given either raster or polygon data as input, a GeoGraph is constructed by assigning each separate patch a graph node. In a second step, edges are added between nodes whenever the patches corresponding to two nodes are within a user-specificed distance. Based on this basic idea, the GeoGraph package provides a wide range of visualisation and analysis tools.

CHAPTER

TWO

WHAT CAN IT BE USED FOR?

Landscape Ecology

Standard Analysis
Building on the graph-based data structure, the GeoGraph package is able to compute most of the standard metrics used in landscape ecology. Combined with an interactive user interface, it provides a powerful Python tool for fragmentation and connectivity analysis.

Policy Advice
Using the tools provided for landscape ecology, the GeoGraph package can be used to give two key insights for policy decisions:

1. Recommend conservation areas
2. Flag areas at potential risk of fragmentation

Temporal Analysis
The graph-based nature of the GeoGraph package allows us to track individual patches over time, and use this information for detailed temporal analysis of habitats.

Polygon Data Visualisation
Whilst our primary use-cases are in landscape ecology, this package can be used to investigate any kind of polygon data files, including `.shp` shape files. The `GeoGraphViewer` allows for the data can be interactively viewed.

2.1 Installation

You can install GeoGraph via pip using

```
pip install geograph
```

2.2 Tutorials

Here we have a number of tutorials introducing the different features of the GeoGraph package. You can also run the tutorials just in your browser without installing anything locally, by using binder with this link.

2.2.1 Basic Usage

Given a variable `data` that is one of

- a path to a pickle file or compressed pickle file to load the graph from,
- a path to vector data in GPKG or Shapefile format,
- a path to raster data in GeoTiff format,
- a numpy array containing raster data,
- a dataframe containing polygons,

you can create a `GeoGraph` using

```
from geograph import GeoGraph
graph = GeoGraph(data)
```

To visualise this graph use the following code in a jupyter notebook

```
from geograph.visualisation.geoviewer import GeoGraphViewer
viewer = GeoGraphViewer()
viewer.add_graph(graph, name='my_graph')
viewer.enable_graph_controls()
viewer
```

This should then look something like

2.2.2 Computing Common Landscape Metrics

This tutorial shows how to compute landscape level metrics using the `GeoGraph` package, and how these compare to `PyLandStats` metrics.

1. Imports

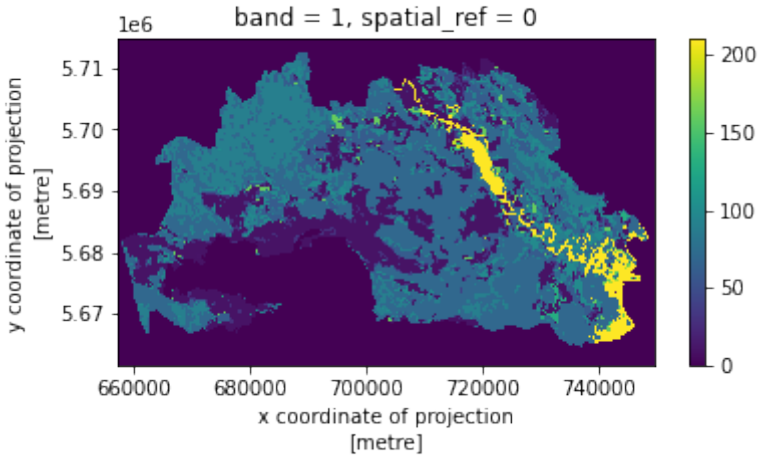
[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import rioxarray as rxr
import geopandas as gpd
import pylandstats as pls
from geograph import GeoGraph
from geograph.constants import UTM35N
from geograph.demo.binder_constants import DATA_DIR, ROIS

# Parse geotif landcover data
chernobyl_path = (
    lambda year: DATA_DIR / "chernobyl" / "esa_cci" / f"esa_cci_{year}_chernobyl.tif"
)

# Parse ROIS
rois = gpd.read_file(ROIS)
cez = rois[rois["name"] == "Chernobyl Exclusion Zone"]
```

2. Loading Chernobyl Landcover Data (ESA CCI)

For this tutorial we will use the ESA CCI landcover dataset as an example to illustrate how geographs enable us to reproduce the same metrics that traditional fragmentation software (pylandstats, fragstats) produce. In this demo we will look specifically at the Chernobyl exclusion zone, so we reproject the satellite data to the right coordinate system (UTM35N) and clip to the CEZ region of interest).

[3]:
```
def clip_and_reproject(xrdata, clip_geometry=None, to_crs=UTM35N, x_res=300, y_
↪res=300):

    if clip_geometry is not None:
        clipped_data = xrdata.rio.clip(clip_geometry)
    else:
        clipped_data = xrdata

    if to_crs is not None:
        reprojected_data = clipped_data.rio.reproject(to_crs, resolution=(x_res, y_
↪res))
    else:
        reprojected_data = clipped_data

    return reprojected_data
```

[ ]:
```
# Loading raster data
cez_raster_2015 = clip_and_reproject(
    rxr.open_rasterio(chernobyl_path(2015)), clip_geometry=cez.geometry
)

# Plot the data
fig, ax = plt.subplots(1, figsize=(6, 3))
cez_raster_2015.plot(cmap="viridis", ax=ax);
```



nbsphinx-code-borderwhite

[5]:
```
# Load geograph from the raster data (construction takes ~10s)
cez_graph_2015 = GeoGraph(
    data=cez_raster_2015.data,
    transform=cez_raster_2015.rio.transform(),
    mask=cez_raster_2015.data > 0,
    crs=UTM35N,
    connectivity=8,
)

# Visualize geograph
```

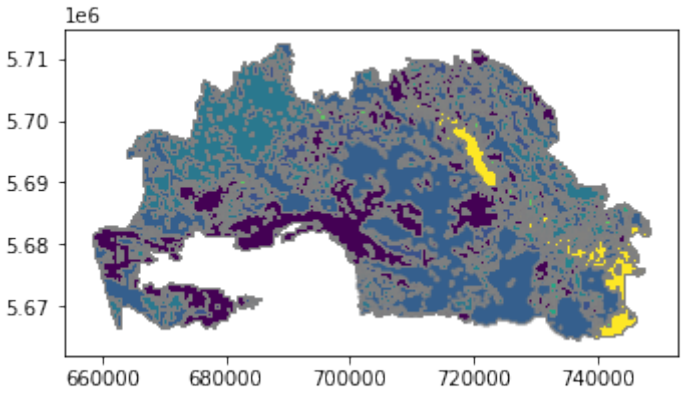
(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1, figsize=(6, 3))
cez_graph_2015.df.plot("class_label", ax=ax, edgecolor="grey");
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1999/1999 [00:01<00:00,␣
↪1147.76it/s]
Step 2 of 2: Adding edges: 100%|| 1999/1999 [00:00<00:00, 64434.03it/s]
```

```
Graph successfully loaded with 1999 nodes and 5117 edges.
```



nbsphinx-code-borderwhite

Next, we can calculate the metrics for this GeoGraph.

3. Investigating metrics

Let us now investigate a couple of the standard metrics that are used in landscape ecology. By default GeoGraph always uses the metrics of the CRS system that the data is in, so meters in our case of UTM35N.

Conveniently, GeoGraph includes the units for us under the "unit" parameter.

3.1 Landscape level metrics

[6]:
```
# Calculating the landscape total area
cez_graph_2015.get_metric("total_area", class_value=11)
```

[6]:
```
Metric(value=21780000.0, name='total_area_class=11', description='Total area of all␣
↪patches of class 11 in the graph.', variant='conventional', unit='CRS.unit**2')
```

[7]:
```
# Calculating the landscape shannon index
cez_graph_2015.get_metric("shannon_diversity_index")
```

[7]:
```
Metric(value=1.7805292346274288, name='shannon_diversity_index', description='SHDI␣
↪approaches 0 when the entire landscape consists of a single patch, and increases as␣
↪the number of classes increases and/or the proportional distribution of area among␣
↪classes becomes more equitable.', variant='conventional', unit='dimensionless')
```

[8]:
```
# Calculating the landscape simpson diversity index
cez_graph_2015.get_metric("simpson_diversity_index")
```

[8]:
```
Metric(value=0.7753989062411218, name='simpson_diversity_index', description=
↪'Probability that any two pixels drawn at random will be of differentclass types',␣
↪variant='conventional', unit='dimensionless')
```

3.1.1 Comparison to pylandstats landscape level metrics

To convince us that these metrics are sensible, we compare to the python implementation pylandstats of the popular FRAGSTATS package. As we will see, GeoGraphs allow us to compute (almost - currently only limited by our time for implementing them) any metric that pylandstas supports. We will also see that the metrics agree, if we note the Caveat that pylandtats uses hecatres as base unit for computations, while GeoGraph uses the unit of the CRS of the underlying data (meters in our case of `UTM35N`).

[9]:
```
cez_landscape_2015 = pls.Landscape(
    cez_raster_2015.data.squeeze(), res=(300, 300), nodata=0
)
```

[10]:
```
print("Shannon diversity index:")
print(f"\t Pylandstats: {cez_landscape_2015.shannon_diversity_index()}")
print(f"\t Geograph:    {cez_graph_2015.get_metric('shannon_diversity_index').value}")
```

```
Shannon diversity index:
         Pylandstats: 1.7805292346274288
         Geograph:    1.7805292346274288
```

[11]:
```
print("Total area:")
print(f"\t Pylandstats: {cez_landscape_2015.total_area()} ha")
print(f"\t Geograph:    {cez_graph_2015.get_metric('total_area').value} meter^2")
```

```
Total area:
         Pylandstats: 259758.0 ha
         Geograph:    2597580000.0 meter^2
```

3.2 Class level metrics

On top of landscape level metrics, we might be interested in how different landcover classes are distributed across the landscape. For this purpose we can use class level metrics:

For a GeoGraph, we can request the metrics individually via the `get_metrics` method as before. If we want to have several metrics for multiple classes, we can also take the `get_class_metrics` shortcut. The GeoGraph caches the computations under the hood to speed up future computations.

[12]:
```
cez_graph_2015.get_metric("proportion_of_landscape", class_value=10)
```

[12]:
```
Metric(value=0.20178781789203798, name='proportion_of_landscape_class=10',␣
↪description='The proportional abundance of 10 in the graph.', variant='conventional
↪', unit='dimensionless')
```

[13]:
```
cez_graph_2015.get_class_metrics(
    names=["num_patches", "effective_mesh_size"], classes=[10, 11, 210]
)
```

[13]:

	num_patches	effective_mesh_size
10	190	2.442215e+07
11	84	7.109694e+03
210	34	1.733824e+06

To get all metrics, simply omit the arguments.

[14]:
```
cez_graph_2015.get_class_metrics()
```

[14]:

	num_patches	avg_patch_area	total_area	proportion_of_landscape	\
10	190	2.758737e+06	524160000.0	0.201788	

(continues on next page)

(continued from previous page)

```
11            84    2.592857e+05   21780000.0                 0.008385
30           274    2.614599e+05   71640000.0                 0.027580
40           131    1.731298e+05   22680000.0                 0.008731
60           243    8.181481e+05  198810000.0                 0.076537
61             3    1.500000e+05     450000.0                 0.000173
70           232    4.201293e+06  974700000.0                 0.375234
90           240    1.855500e+06  445320000.0                 0.171436
100          401    4.607731e+05  184770000.0                 0.071132
130           40    3.397500e+05   13590000.0                 0.005232
150            1    3.600000e+05     360000.0                 0.000139
160          103    2.568932e+05   26460000.0                 0.010186
180           15    1.500000e+05    2250000.0                 0.000866
190            7    3.214286e+05    2250000.0                 0.000866
200            1    1.800000e+05     180000.0                 0.000069
210           34    3.181765e+06  108180000.0                 0.041646

     patch_density  largest_patch_index  total_edge  edge_density  \
10    7.314500e-08             0.093029   1771800.0      0.003380
11    3.233779e-08             0.001213    189000.0      0.008678
30    1.054828e-07             0.001317    668400.0      0.009330
40    5.043156e-08             0.000277    235200.0      0.010370
60    9.354861e-08             0.016977   1102200.0      0.005544
61    1.154921e-09             0.000104      5400.0      0.012000
70    8.931390e-08             0.211143   2670000.0      0.002739
90    9.239369e-08             0.088559   1727400.0      0.003879
100   1.543745e-07             0.002876   1410600.0      0.007634
130   1.539895e-08             0.000832    106200.0      0.007815
150   3.849737e-10             0.000139      2400.0      0.006667
160   3.965229e-08             0.001005    229200.0      0.008662
180   5.774606e-09             0.000173     25200.0      0.011200
190   2.694816e-09             0.000554     19200.0      0.008533
200   3.849737e-10             0.000069      1800.0      0.010000
210   1.308911e-08             0.022694    429600.0      0.003971

     shape_index  effective_mesh_size
10     19.347407         2.442215e+07
11     10.124483         7.109694e+03
30     19.742342         1.646456e+04
40     12.346839         2.195274e+03
60     19.542553         9.896539e+05
61      2.012461         3.430116e+01
70     21.380398         1.195824e+08
90     20.464297         2.110435e+07
100    25.943476         1.386359e+05
130     7.202028         5.628508e+03
150     1.000000         4.989259e+01
160    11.139346         6.666898e+03
180     4.200000         1.964521e+02
190     3.200000         8.637655e+02
200     1.060660         1.247315e+01
210    10.325968         1.733824e+06
```

Let's compare with pylandstats to check the correctness of the metrics:

[15]:
```
pls_class_metrics = [
    "number_of_patches",
```

(continues on next page)

(continued from previous page)

```
    "area_mn",
    "total_area",
    "proportion_of_landscape",
    "patch_density",
    "largest_patch_index",
    "effective_mesh_size",
]

pls_metrics = cez_landscape_2015.compute_class_metrics_df(
    metrics=pls_class_metrics, classes=cez_graph_2015.classes
)
pls_metrics
```

[15]:

class_val	number_of_patches	area_mn	total_area	proportion_of_landscape	patch_density	largest_patch_index	effective_mesh_size
10	190	275.873684	52416.0	20.178782	0.073145	9.302890	2442.214954
11	84	25.928571	2178.0	0.838473	0.032338	0.121267	0.710969
30	274	26.145985	7164.0	2.757952	0.105483	0.131661	1.646456
40	131	17.312977	2268.0	0.873120	0.050432	0.027718	0.219527
60	243	81.814815	19881.0	7.653662	0.093549	1.697734	98.965387
61	3	15.000000	45.0	0.017324	0.001155	0.010394	0.003430
70	232	420.129310	97470.0	37.523387	0.089314	21.114268	11958.236574
90	240	185.550000	44532.0	17.143649	0.092394	8.855935	2110.434828
100	401	46.077307	18477.0	7.113159	0.154374	0.287575	13.863592
130	40	33.975000	1359.0	0.523179	0.015399	0.083154	0.562851
150	1	36.000000	36.0	0.013859	0.000385	0.013859	0.004989
160	103	25.689320	2646.0	1.018640	0.039652	0.100478	0.666690
180	15	15.000000	225.0	0.086619	0.005775	0.017324	0.019645
190	7	32.142857	225.0	0.086619	0.002695	0.055436	0.086377
200	1	18.000000	18.0	0.006930	0.000385	0.006930	0.001247
210	34	318.176471	10818.0	4.164646	0.013089	2.269420	173.382371

As we can see, the metrics agree if we account for the conversion between `metres` and `hectares` and `fractions` in GeoGraph versus `percentages` in pylandstats.

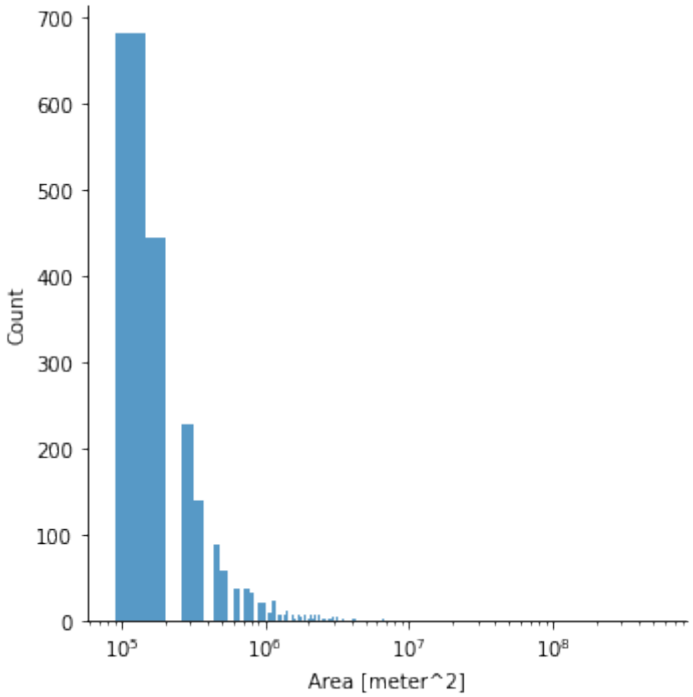
3.2 Patch level metrics

If we're interested in the more fine-grained distribution of metrics on the patch level, GeoGraph can also calculate several standard patch-level metrics of landscape ecology for us.

[16]:
```
patch_metrics = cez_graph_2015.get_patch_metrics()
patch_metrics  # again, metrics are in units of UTM35N (meter)

# Classes of interest
classes_of_interest = patch_metrics["class_label"].isin([10, 11, 210])
```

We can now use this metrics to investigate the underlying patch-area distribution for example

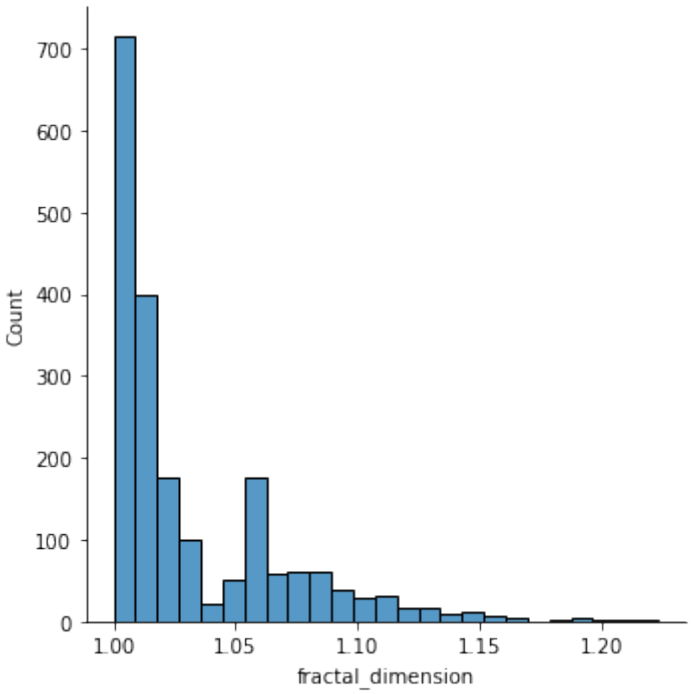
[17]:
```
import seaborn as sns

sns.displot(patch_metrics["area"])
plt.xlabel("Area [meter^2]")
plt.semilogx();
```



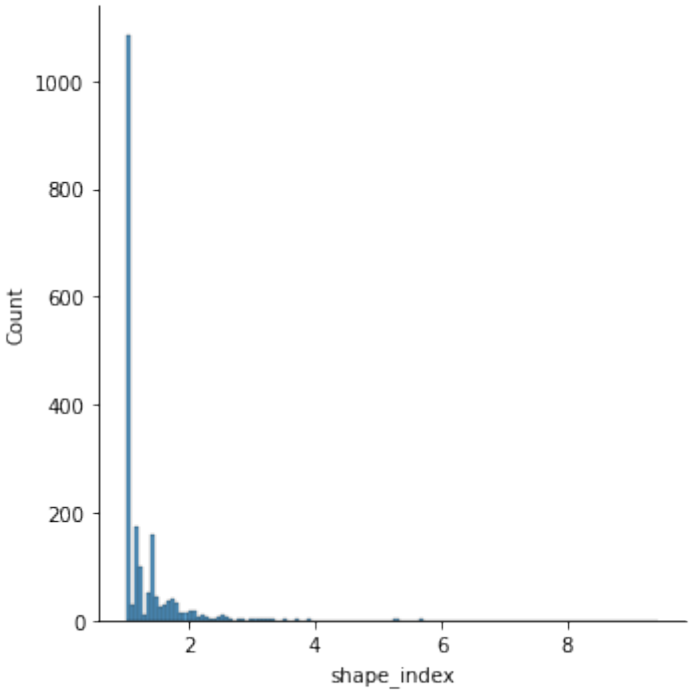
nbsphinx-code-borderwhite

We can also investigate the dimensionless metrics `shape_index`, `fractal_dimension`.

[18]:
```
sns.displot(patch_metrics["fractal_dimension"])
sns.displot(patch_metrics["shape_index"]);
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

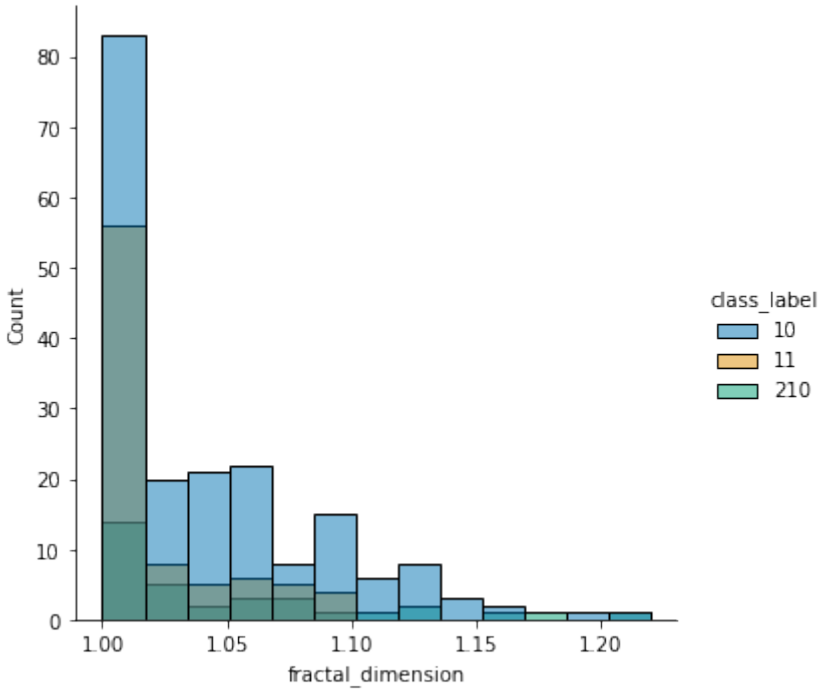
Further, we can easily analyse distributions for certain classes.

[19]:
```
sns.displot(
    patch_metrics[classes_of_interest],
    x="fractal_dimension",
    hue="class_label",
    palette="colorblind",
)
sns.displot(
    patch_metrics[classes_of_interest],
    x="perimeter_area_ratio",
    hue="class_label",
```

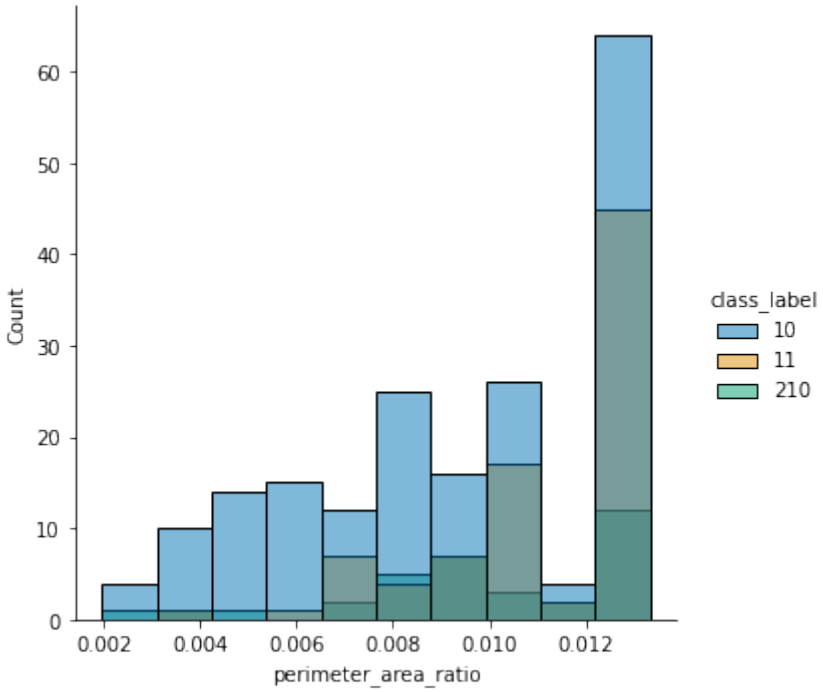
(continues on next page)

(continued from previous page)

```
    palette="colorblind",
);
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

2.2.3 Computing Landscape Metrics over Time

This tutorial shows how to use the `GeoGraph` package to compute landscape metrics over time.

1. Imports

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import pyplot
import seaborn as sns
import rioxarray as rxr
import geopandas as gpd
import pylandstats as pls
from geograph import GeoGraph
from geograph.geotimeline import GeoGraphTimeline
from geograph.constants import UTM35N
from geograph.demo.binder_constants import DATA_DIR, ROIS
from geograph.demo.plot_settings import ps_defaults, label_subplots, set_dim

ps_defaults(use_tex=True)
```

2. Loading Chernobyl Landcover data (ESA CCI)

For this demonstration we will use the ESA CCI landcover dataset as an example to illustrate how geographs enable us to reproduce the same metrics that traditional fragmentation software (pylandstats, fragstats) produce.

In this demo we will look specifically at the Chernobyl exclusion zone, so we reproject the satellite data to the right coordinate system (UTM35N) and clip to the CEZ region of interest).

[3]:
```
# Parse geotif landcover data
chernobyl_path = (
    lambda year: DATA_DIR / "chernobyl" / "esa_cci" / f"esa_cci_{year}_chernobyl.tif"
)

# Parse ROIS
rois = gpd.read_file(ROIS)
cez = rois[rois["name"] == "Chernobyl Exclusion Zone"]
pez = rois[rois["name"] == "Polesie Exclusion Zone"]
e30 = rois[rois["name"] == "E+30"]
e60 = rois[rois["name"] == "E+60"]
```

For convenience, let us also define certain groups of land cover classes from the ESA CCI data (c.f. Appendix of this notebook for a full list of classes)

[4]:
```
cropland = [10, 20]
deciduous_forest = [60, 61, 62, 80, 81, 82]
evergreen_forest = [70, 71, 72]
wetlands = [160, 170, 180]
urban = [190]
water = [210]
```

Next, let us load the raster data into memory, clip to the regions of interest and reproject to UTM35N

[5]:
```
def clip_and_reproject(xrdata, clip_geometry=None, to_crs=UTM35N, x_res=300, y_
↪res=300):

    if clip_geometry is not None:
        clipped_data = xrdata.rio.clip(clip_geometry)
    else:
        clipped_data = xrdata

    if to_crs is not None:
        reprojected_data = clipped_data.rio.reproject(to_crs, resolution=(x_res, y_
↪res))
    else:
        reprojected_data = clipped_data

    return reprojected_data
```

[6]:
```
# Loading raster data (for pylandstats and graph creation)
years = range(2000, 2015)
zones = [cez, pez, e30, e60]


rasters = {
    zone.name.iloc[0]: {
        year: clip_and_reproject(
            rxr.open_rasterio(chernobyl_path(year)), clip_geometry=zone.geometry
        )
        for year in years
    }
    for zone in zones
}
```

2.1 Loading the Chernobyl Exclusion Zone (CEZ)

Next, let us load the geographs for the chernobyl exclusion zone and its surroundings for all years from 2000 to 2014. To save you some time, we have pre-computed them and stored them already, so we can simply load them from disk.

If you would nontheless like to recreate them from the original raster data, which we loaded above, the code to do so is commented below. Simply uncomment and run to recreate the graphs from scratch.

[7]:
```
# Demo path includes pre-loaded graphs for faster loading
demo_path = DATA_DIR / "chernobyl" / "graphs"
```

[8]:
```
# Loading pre-created geographs from disk
cez_graphs = {year: GeoGraph(demo_path / f"cez_graph_{year}.gz") for year in years}

## To recreate the graphs from scratch, run this: (~2-3s per graph)
# cez_graphs = {}
# for year, raster in rasters["Chernobyl Exclusion Zone"].items():
#     print(f"Analysing year {year}")
#     cez_graphs[year] = GeoGraph(data=raster.data.squeeze(),
#                                 transform=raster.rio.transform(),
#                                 mask=raster.data.squeeze() > 0,
#                                 crs=UTM35N,
#                                 connectivity=8)
```

```
Graph successfully loaded with 1924 nodes and 4912 edges.
Graph successfully loaded with 1931 nodes and 4918 edges.
Graph successfully loaded with 1929 nodes and 4897 edges.
Graph successfully loaded with 1936 nodes and 4911 edges.
Graph successfully loaded with 1953 nodes and 4953 edges.
Graph successfully loaded with 1960 nodes and 4973 edges.
Graph successfully loaded with 2004 nodes and 5113 edges.
Graph successfully loaded with 1996 nodes and 5141 edges.
Graph successfully loaded with 1992 nodes and 5119 edges.
Graph successfully loaded with 1994 nodes and 5108 edges.
Graph successfully loaded with 1988 nodes and 5080 edges.
Graph successfully loaded with 2003 nodes and 5131 edges.
Graph successfully loaded with 1998 nodes and 5119 edges.
Graph successfully loaded with 2003 nodes and 5140 edges.
Graph successfully loaded with 1999 nodes and 5117 edges.
```

[9]:
```python
# Loading raster data into pylandstats
cez_landscapes = {
    year: pls.Landscape(
        rasters["Chernobyl Exclusion Zone"][year].data.squeeze(),
        res=(300, 300),
        nodata=0,
    )
    for year in years
}
```

2.2 Loading the Polesia Exclusion Zone (PEZ)

[10]:
```python
# Loading pre-created geographs from disk
pez_graphs = {year: GeoGraph(demo_path / f"pez_graph_{year}.gz") for year in years}

## To recreate the graphs from scratch, run this: (~2-3s per graph)
# pez_graphs = {}
# for year, raster in rasters["Polesie Exclusion Zone"].items():
#     print(f"Analysing year {year}")
#     pez_graphs[year] = GeoGraph(data=raster.data.squeeze(),
#                                 transform=raster.rio.transform(),
#                                 mask=raster.data.squeeze() > 0,
#                                 crs=UTM35N,
#                                 connectivity=8)
```

```
Graph successfully loaded with 1942 nodes and 5223 edges.
Graph successfully loaded with 1940 nodes and 5222 edges.
Graph successfully loaded with 1940 nodes and 5227 edges.
Graph successfully loaded with 1933 nodes and 5210 edges.
Graph successfully loaded with 1931 nodes and 5201 edges.
Graph successfully loaded with 1936 nodes and 5209 edges.
Graph successfully loaded with 1937 nodes and 5215 edges.
Graph successfully loaded with 1932 nodes and 5184 edges.
Graph successfully loaded with 1920 nodes and 5152 edges.
Graph successfully loaded with 1905 nodes and 5099 edges.
Graph successfully loaded with 1908 nodes and 5098 edges.
Graph successfully loaded with 1907 nodes and 5091 edges.
Graph successfully loaded with 1907 nodes and 5094 edges.
Graph successfully loaded with 1912 nodes and 5111 edges.
Graph successfully loaded with 1924 nodes and 5147 edges.
```

[11]:
```
# Loading raster data into pylandstats
pez_landscapes = {
    year: pls.Landscape(
        rasters["Polesie Exclusion Zone"][year].data.squeeze(), res=(300, 300),␣
↪nodata=0
    )
    for year in years
}
```

2.3 Loading the 30 km surroundings

[12]:
```
# Loading pre-created geographs from disk
e30_graphs = {year: GeoGraph(demo_path / f"e30_graph_{year}.gz") for year in years}

## To recreate the graphs from scratch, run this: (~25s per graph)
# e30_graphs = {}
# for year, raster in rasters["E+30"].items():
#     print(f"Analysing year {year}")
#     e30_graphs[year] = GeoGraph(data=raster.data.squeeze(),
#                                 transform=raster.rio.transform(),
#                                 mask=raster.data.squeeze() > 0,
#                                 crs=UTM35N,
#                                 connectivity=8)
```

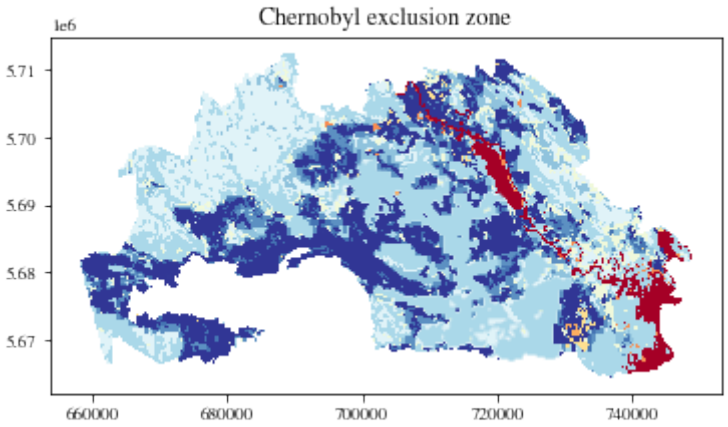
```
Graph successfully loaded with 8468 nodes and 20612 edges.
Graph successfully loaded with 8484 nodes and 20644 edges.
Graph successfully loaded with 8473 nodes and 20595 edges.
Graph successfully loaded with 8478 nodes and 20577 edges.
Graph successfully loaded with 8499 nodes and 20633 edges.
Graph successfully loaded with 8480 nodes and 20547 edges.
Graph successfully loaded with 8500 nodes and 20625 edges.
Graph successfully loaded with 8512 nodes and 20649 edges.
Graph successfully loaded with 8501 nodes and 20604 edges.
Graph successfully loaded with 8485 nodes and 20568 edges.
Graph successfully loaded with 8504 nodes and 20654 edges.
Graph successfully loaded with 8507 nodes and 20649 edges.
Graph successfully loaded with 8507 nodes and 20647 edges.
Graph successfully loaded with 8510 nodes and 20663 edges.
Graph successfully loaded with 8514 nodes and 20658 edges.
```

[13]:
```
# Loading raster data into pylandstats
e30_landscapes = {
    year: pls.Landscape(rasters["E+30"][year].data.squeeze(), res=(300, 300),␣
↪nodata=0)
    for year in years
}
```

2.4 Plot the data

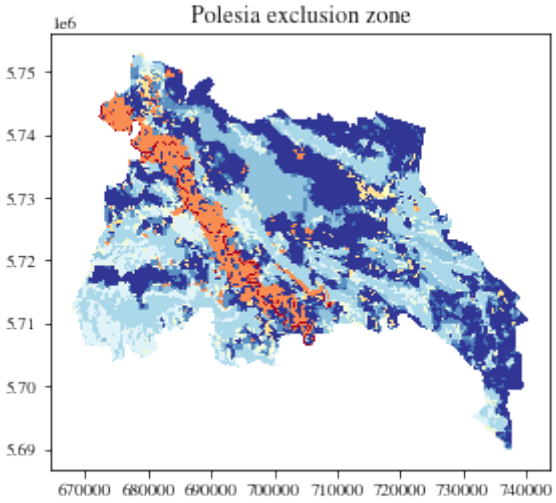
Let's plot a sample region for the year 2000 for each of the regions to see that we correctly loaded the data

[14]:
```
cez_graphs[2000].df.plot("class_label")
plt.title("Chernobyl exclusion zone");
```



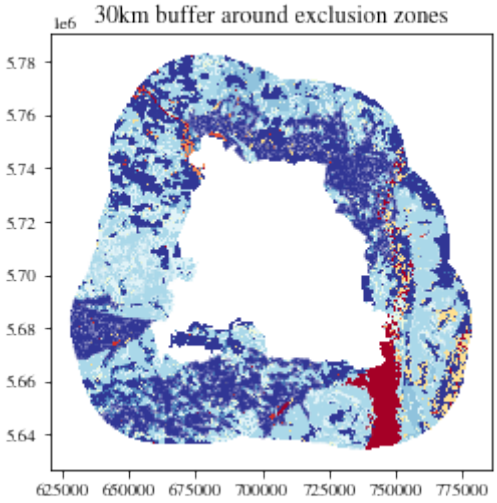
nbsphinx-code-borderwhite

[15]:
```
pez_graphs[2000].df.plot("class_label")
plt.title("Polesia exclusion zone");
```



nbsphinx-code-borderwhite

[16]:
```
e30_graphs[2000].df.plot("class_label")
plt.title("30km buffer around exclusion zones");
```



nbsphinx-code-borderwhite

3. Metrics calculation

[17]:
```
cez_graph_timestack = GeoGraphTimeline(cez_graphs)
pez_graph_timestack = GeoGraphTimeline(pez_graphs)
e30_graph_timestack = GeoGraphTimeline(e30_graphs)
```

[18]:
```
# Calculate pylandstats shannon diversity indices for entire landscapes
cez_pls_shannon = [cez_landscapes[year].shannon_diversity_index() for year in years]
pez_pls_shannon = [pez_landscapes[year].shannon_diversity_index() for year in years]
e30_pls_shannon = [e30_landscapes[year].shannon_diversity_index() for year in years]
```

[19]:
```
# Calculate pylandstats effective mesh size for class 100 (mixed forest and shrub␣
↪lands)
cez_pls_effm = [cez_landscapes[year].effective_mesh_size(100) for year in years]
pez_pls_effm = [pez_landscapes[year].effective_mesh_size(100) for year in years]
e30_pls_effm = [e30_landscapes[year].effective_mesh_size(100) for year in years]
```

[20]:
```
# Calculate pylandstats mean fractal dimension for class 100 (mixed forest and shrub␣
↪lands)
cez_pls_fracdim = [cez_landscapes[year].fractal_dimension_mn(100) for year in years]
pez_pls_fracdim = [pez_landscapes[year].fractal_dimension_mn(100) for year in years]
e30_pls_fracdim = [e30_landscapes[year].fractal_dimension_mn(100) for year in years]
```

[21]:
```
# Calculate time-series patch metrics for geographs
cez_patch_metrics = cez_graph_timestack.get_patch_metrics("mean")
pez_patch_metrics = pez_graph_timestack.get_patch_metrics("mean")
e30_patch_metrics = e30_graph_timestack.get_patch_metrics("mean")

cez_patch_metrics_std = cez_graph_timestack.get_patch_metrics("std")
pez_patch_metrics_std = pez_graph_timestack.get_patch_metrics("std")
e30_patch_metrics_std = e30_graph_timestack.get_patch_metrics("std")
```

[23]:
```
import xarray as xr

# Load pre-computed wetland habitats
wetlands_0km_component_isolation = (
    xr.open_dataset(DATA_DIR / "chernobyl" / "wetlands" / "cez_wetlands_0km.ncdf")
```

(continues on next page)

(continued from previous page)

```
    .to_array()
    .squeeze()
)
wetlands_1km_component_isolation = (
    xr.open_dataset(DATA_DIR / "chernobyl" / "wetlands" / "cez_wetlands_1km.ncdf")
    .to_array()
    .squeeze()
)
wetlands_5km_component_isolation = (
    xr.open_dataset(DATA_DIR / "chernobyl" / "wetlands" / "cez_wetlands_5km.ncdf")
    .to_array()
    .squeeze()
)
wetlands_10km_component_isolation = (
    xr.open_dataset(DATA_DIR / "chernobyl" / "wetlands" / "cez_wetlands_10km.ncdf")
    .to_array()
    .squeeze()
)
wetlands_20km_component_isolation = (
    xr.open_dataset(DATA_DIR / "chernobyl" / "wetlands" / "cez_wetlands_20km.ncdf")
    .to_array()
    .squeeze()
)

# Warning - computation of all edges takes a long time (~tens of minutes)

# cez_graph_timestack.add_habitat("wetlands (0 km)", valid_classes=wetlands, max_
↪travel_distance=0)
# cez_graph_timestack.habitats["wetlands (0 km)"].get_metric("avg_component_isolation
↪")
# cez_graph_timestack.habitats["wetlands (0 km)"].get_metric("avg_component_isolation
↪").to_netcdf("./cez_wetlands_0km.ncdf")

# cez_graph_timestack.add_habitat("wetlands (1 km)", valid_classes=wetlands, max_
↪travel_distance=1e3)
# cez_graph_timestack.habitats["wetlands (1 km)"].get_metric("avg_component_isolation
↪")
# cez_graph_timestack.habitats["wetlands (1 km)"].get_metric("avg_component_isolation
↪").to_netcdf("./cez_wetlands_1km.ncdf")

# cez_graph_timestack.add_habitat("wetlands (5 km)", valid_classes=wetlands, max_
↪travel_distance=5e3)
# cez_graph_timestack.habitats["wetlands (5 km)"].get_metric("avg_component_isolation
↪")
# cez_graph_timestack.habitats["wetlands (5 km)"].get_metric("avg_component_isolation
↪").to_netcdf("./cez_wetlands_5km.ncdf")

# cez_graph_timestack.add_habitat("wetlands (10 km)", valid_classes=wetlands, max_
↪travel_distance=10e3)
# cez_graph_timestack.habitats["wetlands (10 km)"].get_metric("avg_component_isolation
↪")
# cez_graph_timestack.habitats["wetlands (10 km)"].get_metric("avg_component_isolation
↪").to_netcdf("./cez_wetlands_10km.ncdf")

# cez_graph_timestack.add_habitat("wetlands (20 km)", valid_classes=wetlands, max_
↪travel_distance=20e3)
```

(continues on next page)

(continued from previous page)

```
# cez_graph_timestack.habitats["wetlands (20 km)"].get_metric("avg_component_isolation
↪")
# cez_graph_timestack.habitats["wetlands (20 km)"].get_metric("avg_component_isolation
↪").to_netcdf("./cez_wetlands_20km.ncdf")
```

[25]:
```
# Helper function with boilerplate code for plotting graph metrics
plot_scale_factor = 1.8  # To scale plot sizes for report


def plot_graph_metrics(
    ax,
    metric,
    class_value=None,
    plot_scale_factor=plot_scale_factor,
    marker="*",
    linewidth=1,
    conversion_factor=1,
    markersize=3,
):

    ax.plot(
        cez_graph_timestack.times,
        cez_graph_timestack.get_metric(metric, class_value) * conversion_factor,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C0",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )

    ax.plot(
        pez_graph_timestack.times,
        pez_graph_timestack.get_metric(metric, class_value) * conversion_factor,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C1",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )

    ax.plot(
        e30_graph_timestack.times,
        e30_graph_timestack.get_metric(metric, class_value) * conversion_factor,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C2",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )
```

[26]:
```
# Helper function with boilerplate code for plotting pylandstats metrics
def plot_pls_metric(
    ax,
    metrics,
    plot_scale_factor=plot_scale_factor,
    marker="x",
```

(continues on next page)

(continued from previous page)

```
    linewidth=1,
    markersize=4,
):

    cez_vals, pez_vals, e30_vals = metrics

    ax.plot(
        cez_graph_timestack.times,
        cez_vals,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C0",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )

    ax.plot(
        pez_graph_timestack.times,
        pez_vals,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C1",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )

    ax.plot(
        e30_graph_timestack.times,
        e30_vals,
        linewidth=linewidth * plot_scale_factor,
        linestyle="dashed",
        color="C2",
        marker=marker,
        markersize=markersize * plot_scale_factor,
    )
```

[27]:
```
colors = sns.color_palette("rocket_r").as_hex()
colors
```

[27]:
```
['#f6b48f', '#f37651', '#e13342', '#ad1759', '#701f57', '#35193e']
```

[28]:
```
fig, ax = plt.subplots(2, 2, sharex="col")
set_dim(fig, fraction_of_line_width=plot_scale_factor)
colors = sns.color_palette("rocket_r").as_hex()

# Top left plot
ax[0, 0].set_title(
    "Landscape:\nShannon Diversity Index", fontsize=8 * plot_scale_factor
)
plot_graph_metrics(
    ax[0, 0],
    "shannon_diversity_index",
    marker="o",
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=2,
```

(continues on next page)

(continued from previous page)

```
)
plot_pls_metric(
    ax[0, 0],
    metrics=[cez_pls_shannon, pez_pls_shannon, e30_pls_shannon],
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=3,
)

# Top right plot
ax[0, 1].set_title(
    "Mosaic tree and shrub:\nEffective mesh size [ha]", fontsize=8 * plot_scale_factor
)
plot_graph_metrics(
    ax[0, 1],
    "effective_mesh_size",
    class_value=100,
    conversion_factor=0.0001,
    marker="o",
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=2,
)
plot_pls_metric(
    ax[0, 1],
    [cez_pls_effm, pez_pls_effm, e30_pls_effm],
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=3,
)

# Bottom left plot
ax[1, 0].set_title(
    "Mosaic tree and shrub:\nAvg. Fractal Dimension", fontsize=8 * plot_scale_factor
)
plot_pls_metric(
    ax[1, 0],
    [
        cez_patch_metrics.loc[:, 100, "fractal_dimension"],
        pez_patch_metrics.loc[:, 100, "fractal_dimension"],
        e30_patch_metrics.loc[:, 100, "fractal_dimension"],
    ],
    marker="o",
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=2,
)
plot_pls_metric(
    ax[1, 0],
    [cez_pls_fracdim, pez_pls_fracdim, e30_pls_fracdim],
    plot_scale_factor=plot_scale_factor,
    linewidth=0.5,
    markersize=3,
)

markersize = 2
```

(continues on next page)

(continued from previous page)

```
ax[1, 1].set_title(
    "CEZ Wetland:\nAvg. Component Isolation [km]", fontsize=8 * plot_scale_factor
)
ax[1, 1].plot(
    cez_graph_timestack.times,
    wetlands_0km_component_isolation / 1e3,
    linewidth=1 * plot_scale_factor,
    linestyle="dotted",
    color=colors[5],
    marker="o",
    markersize=markersize * plot_scale_factor,
)

ax[1, 1].plot(
    cez_graph_timestack.times,
    wetlands_1km_component_isolation / 1e3,
    linewidth=1 * plot_scale_factor,
    linestyle="dotted",
    color=colors[4],
    marker="o",
    markersize=markersize * plot_scale_factor,
)

ax[1, 1].plot(
    cez_graph_timestack.times,
    wetlands_5km_component_isolation / 1e3,
    linewidth=1 * plot_scale_factor,
    linestyle="dotted",
    color=colors[2],
    marker="o",
    markersize=markersize * plot_scale_factor,
)

ax[1, 1].plot(
    cez_graph_timestack.times,
    wetlands_10km_component_isolation / 1e3,
    linewidth=1 * plot_scale_factor,
    linestyle="dotted",
    color=colors[1],
    marker="o",
    markersize=markersize * plot_scale_factor,
)

ax[1, 1].plot(
    cez_graph_timestack.times,
    wetlands_20km_component_isolation / 1e3,
    linewidth=1 * plot_scale_factor,
    linestyle="dotted",
    color=colors[0],
    marker="o",
    markersize=markersize * plot_scale_factor,
)


# Set labels
ax[0, 0].set_xticks([])
```

(continues on next page)

(continued from previous page)

```
ax[0, 0].tick_params(axis="both", labelsize=8 * plot_scale_factor)
ax[0, 1].tick_params(axis="both", labelsize=8 * plot_scale_factor)
ax[1, 0].tick_params(axis="both", labelsize=8 * plot_scale_factor)

label_years = [2001, 2005, 2009, 2013]
ax[1, 0].set_xticks(label_years)
ax[1, 1].set_xticks(label_years)
ax[1, 0].set_xticklabels(label_years, fontsize=8 * plot_scale_factor)
ax[1, 1].set_xticklabels(label_years, fontsize=8 * plot_scale_factor)
# ax[0,0].stick_params(fontsize=9*plot_scale_factor)
plt.yticks(fontsize=9 * plot_scale_factor)
# ax[0,0].ylabel("Shannon diversity index", fontsize=10*plot_scale_factor)
# ax[0,0].xlabel("Year", fontsize=10*plot_scale_factor)

sns.despine()

from matplotlib.patches import Patch
from matplotlib.lines import Line2D

legend_elements = [
    Line2D(
        [0], [0], color="C0", label="Chernobyl EZ", linewidth=1.5 * plot_scale_factor
    ),
    Line2D([0], [0], color="C1", label="Polesia EZ", linewidth=1.5 * plot_scale_
↪factor),
    Line2D(
        [0], [0], color="C2", label="30 km Buffer", linewidth=1.5 * plot_scale_factor
    ),
    Line2D(
        [0],
        [0],
        marker="o",
        color="grey",
        label="GeoGraph",
        markerfacecolor="grey",
        markersize=5 * plot_scale_factor,
    ),
    Line2D(
        [0],
        [0],
        marker="x",
        color="grey",
        label="Pylandstats",
        markerfacecolor="grey",
        markersize=5 * plot_scale_factor,
    ),
]

fig.legend(
    handles=legend_elements,
    loc="lower left",
    fontsize=9 * plot_scale_factor,
    ncol=2,
    bbox_to_anchor=(0.08, -0.2),
)
```

(continues on next page)

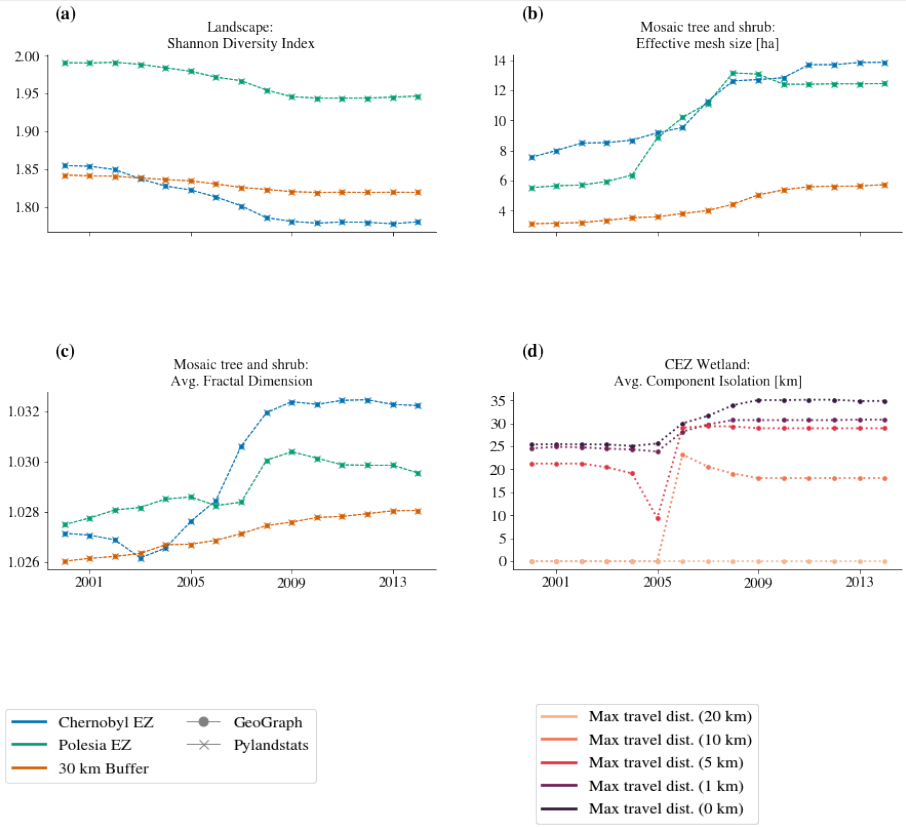
(continued from previous page)

```
legend_elements2 = [
    Line2D(
        [0],
        [0],
        color=colors[0],
        label="Max travel dist. (20 km)",
        linestyle="solid",
        linewidth=1.5 * plot_scale_factor,
    ),
    Line2D(
        [0],
        [0],
        color=colors[1],
        label="Max travel dist. (10 km)",
        linestyle="solid",
        linewidth=1.5 * plot_scale_factor,
    ),
    Line2D(
        [0],
        [0],
        color=colors[2],
        label="Max travel dist. (5 km)",
        linestyle="solid",
        linewidth=1.5 * plot_scale_factor,
    ),
    Line2D(
        [0],
        [0],
        color=colors[4],
        label="Max travel dist. (1 km)",
        linestyle="solid",
        linewidth=1.5 * plot_scale_factor,
    ),
    Line2D(
        [0],
        [0],
        color=colors[5],
        label="Max travel dist. (0 km)",
        linestyle="solid",
        linewidth=1.5 * plot_scale_factor,
    ),
]

legend1 = fig.legend(
    handles=legend_elements2,
    loc="lower left",
    fontsize=9 * plot_scale_factor,
    ncol=1,
    bbox_to_anchor=(0.56, -0.26),
)
pyplot.gca().add_artist(legend1)

inset_text = ""

# these are matplotlib.patch.Patch properties
props = dict(boxstyle="round", facecolor="white", alpha=0.8)
```

(continues on next page)

(continued from previous page)

```
# place a text box in upper left in axes coords
plt.subplots_adjust(hspace=0.5 * plot_scale_factor)
label_subplots(
    axs=ax, labels=["a", "b", "c", "d"], fontsize=10 * plot_scale_factor, y_pos=1.27
)

plt.savefig("CEZ_levels_timestack_analysis.svg", bbox_inches="tight")
plt.savefig("CEZ_levels_timestack_analysis.pdf", bbox_inches="tight")
# plt.savefig("CEZ_levels_timestack_analysis.png")
plt.show()
```



nbsphinx-code-borderwhite

A. Appendix:

Legend of the ESA CCI landcover classes

Legend of the global CCI-LC maps, based on LCCS

Value	Label	Color
0	No Data	
10	Cropland, rainfed	
11	Herbaceous cover	
12	Tree or shrub cover	
20	Cropland, irrigated or post-flooding	
30	Mosaic cropland (>50%) / natural vegetation (tree, shrub, herbaceous cover) (<50%)	
40	Mosaic natural vegetation (tree, shrub, herbaceous cover) (>50%) / cropland (<50%)	
50	Tree cover, broadleaved, evergreen, closed to open (>15%)	
60	Tree cover, broadleaved, deciduous, closed to open (>15%)	
61	Tree cover, broadleaved, deciduous, closed (>40%)	
62	Tree cover, broadleaved, deciduous, open (15-40%)	
70	Tree cover, needleleaved, evergreen, closed to open (>15%)	
71	Tree cover, needleleaved, evergreen, closed (>40%)	
72	Tree cover, needleleaved, evergreen, open (15-40%)	
80	Tree cover, needleleaved, deciduous, closed to open (>15%)	
81	Tree cover, needleleaved, deciduous, closed (>40%)	
82	Tree cover, needleleaved, deciduous, open (15-40%)	
90	Tree cover, mixed leaf type (broadleaved and needleleaved)	
100	Mosaic tree and shrub (>50%) / herbaceous cover (<50%)	
110	Mosaic herbaceous cover (>50%) / tree and shrub (<50%)	
120	Shrubland	
121	Evergreen shrubland	
122	Deciduous shrubland	
130	Grassland	
140	Lichens and mosses	
150	Sparse vegetation (tree, shrub, herbaceous cover) (<15%)	
151	Sparse tree (<15%)	
152	Sparse shrub (<15%)	
153	Sparse herbaceous cover (<15%)	
160	Tree cover, flooded, fresh or brakish water	
170	Tree cover, flooded, saline water	
180	Shrub or herbaceous cover, flooded, fresh/saline/brakish water	
190	Urban areas	
200	Bare areas	
201	Consolidated bare areas	
202	Unconsolidated bare areas	
210	Water bodies	
220	Permanent snow and ice	

2.2.4 Visualising GeoGraph Interactively

This tutorial shows how to visualise a GeoGraph on an interactive map.

---

1. Setup and Loading package

[1]:
```
import ipyleaflet
import geograph
from geograph.visualisation import geoviewer
from geograph.constants import UTM35N
from geograph.demo.binder_constants import DATA_DIR
```

---

2. Loading Data

---

3. Creating GeoGraph

[3]:
```
# Building the main graph structure
graph = geograph.GeoGraph(
    gdf, crs=UTM35N, columns_to_rename={"Eunis_name": "class_label", "AREA": "area"}
)
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 323/323 [00:00<00:00, 569.
↪78it/s]
Step 2 of 2: Adding edges: 100%|| 323/323 [00:00<00:00, 66710.67it/s]
```

```
Graph successfully loaded with 323 nodes and 816 edges.
```

---

4. Creating Habitats

[4]:
```
# First selecting the classes that make up our habitat
# We chose all classes with 'pine' in the name.
pine_classes = [label for label in graph.df.class_label.unique() if "pine" in label]
pine_classes
```

[4]:
```
['Subcontinental moss Scots pine forests',
 'Subcontinental lichen Scots pine forests',
 'Subcontinental moorgrass Scots pine forests',
 'Boreal Labrador tea Scots pine bog woods',
 'Boreal cottonsedge Scots pine bog woods',
 'Boreal neutrocline sphagnum Scots pine fen woods',
 'Mixed Scots pine-birch woodland']
```

[5]:
```
# Distances: mobile (<100m), semi mobile (<25m) and sessile (<5m)
# (proposed by Adham Ashton-Butt at BTO)
graph.add_habitat("Sessile", max_travel_distance=5, valid_classes=pine_classes)

graph.add_habitat("Semi mobile", max_travel_distance=25, valid_classes=pine_classes)

graph.add_habitat("Mobile", max_travel_distance=500, valid_classes=pine_classes)
```

```
Generating habitat graph: 100%|| 95/95 [00:00<00:00, 2926.68it/s]
Constructing graph: 100%|| 39/39 [00:00<00:00, 11990.75it/s]
```

```
Calculating components...
Habitat successfully loaded with 95 nodes and 78 edges.
```

```
Generating habitat graph: 100%|| 95/95 [00:00<00:00, 2645.28it/s]
Constructing graph: 100%|| 36/36 [00:00<00:00, 12607.08it/s]
```

```
Calculating components...
Habitat successfully loaded with 95 nodes and 86 edges.
```

```
Generating habitat graph: 100%|| 95/95 [00:00<00:00, 2717.04it/s]
Constructing graph: 100%|| 14/14 [00:00<00:00, 7885.09it/s]
```

```
Calculating components...
Habitat successfully loaded with 95 nodes and 214 edges.
```

5. Interactive Graph

[6]:
```
viewer = geoviewer.GeoGraphViewer(small_screen=True)
viewer.add_layer(ipyleaflet.basemaps.Esri.WorldImagery)
viewer.add_graph(graph, name="Polesia data", with_components=True)
viewer.enable_graph_controls()
viewer
```

```
Constructing graph: 100%|| 1/1 [00:00<00:00, 1093.69it/s]
Constructing graph: 100%|| 39/39 [00:00<00:00, 9548.09it/s]
Constructing graph: 100%|| 36/36 [00:00<00:00, 10671.77it/s]
Constructing graph: 100%|| 14/14 [00:00<00:00, 7224.44it/s]
```

```
GeoGraphViewer(center=[51.389167, 30.099444], controls=(ZoomControl(options=['position
↪', 'zoom_in_text', 'zoom...
```

Note: an interactive viewer will show up here.

2.2.5 Visualising Temporal Changes in GeoGraphs Interactively

This tutorial shows how to create and visualise a timeline of GeoGraphs.

1. Setup and Loading package

Let us start by installing all relevant dependencies

[2]:
```
import ipyleaflet
import pandas as pd
import geopandas as gpd
import rioxarray as rxr
import geograph
from geograph import geotimeline
from geograph.visualisation import geoviewer
from geograph.constants import UTM35N
from geograph.demo.binder_constants import DATA_DIR, ROIS, ESA_CCI_LEGEND_LINK
from geograph.metrics import LANDSCAPE_METRICS_DICT, COMPONENT_METRICS_DICT
```

```
/home/users/svm/Code/gtc-biodiversity/env/lib/python3.8/site-packages/geopandas/_
↪compat.py:84: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is␣
↪incompatible with the GEOS version PyGEOS was compiled with (3.9.0-CAPI-1.16.2).␣
↪Conversions between both will be slow.
  warnings.warn(
```

---

2. Loading Data

Next, we will load the data for the Chernobyl region. For this example we will use land cover maps from the ESA CCI land cover dataset. Specifically, we will look at the years 2013 and 2014 for the Chernobyl exclusion zone. All data comes pre-installed on the binder in the `DATA_DIR` that we imported from the `binder_constants`. If you are following this demo on your local machine, you can download the data with this link.

[3]:
```
# Parse geotif landcover data
chernobyl_path = (
    lambda year: DATA_DIR / "chernobyl" / "esa_cci" / f"esa_cci_{year}_chernobyl.tif"
)

# Parse ROIS
rois = gpd.read_file(ROIS)
# Load the shape of the chernobyl exclusion zone
cez = rois[rois["name"] == "Chernobyl Exclusion Zone"]
```

[4]:
```
def clip_and_reproject(xrdata, clip_geometry=None, to_crs=UTM35N, x_res=300, y_
↪res=300):

    if clip_geometry is not None:
        clipped_data = xrdata.rio.clip(clip_geometry)
    else:
        clipped_data = xrdata

    if to_crs is not None:
        reprojected_data = clipped_data.rio.reproject(to_crs, resolution=(x_res, y_
↪res))
    else:
        reprojected_data = clipped_data

    return reprojected_data
```

[5]:
```
# Loading ESA CCI land cover raster data
years = list(range(2013, 2015))
cez_rasters = {
    year: clip_and_reproject(
        rxr.open_rasterio(chernobyl_path(year)), clip_geometry=cez.geometry
    )
    for year in years
}
```

[6]:
```
# Loading ESA CCI land cover legend to translate land cover labels to classes
esa_cci_legend = pd.read_csv(ESA_CCI_LEGEND_LINK, delimiter=";", index_col=0)
print(f"There are {len(esa_cci_legend)} classes.")

class_value_to_label = {
    class_val: row.LCCOwnLabel for class_val, row in esa_cci_legend.iterrows()
}
```

```
There are 38 classes.
```

---

3. Creating GeoGraph

[7]:
```
# Polygonising raster and transforming into graph
cez_graphs = {}
for year, raster in cez_rasters.items():
    print(f"Analysing year {year}")
    # Load geograph from the raster data (construction takes ~10s)
    cez_graphs[year] = geograph.GeoGraph(
        data=raster.data,
        transform=raster.rio.transform(),
        crs=UTM35N,
        mask=raster.data > 0,
        connectivity=8,
    )
    # Map the ESA CCI land cover class value (int) to the plain text label
    #  to have the plain text labels available in the interactive viewer.
    cez_graphs[year].df.class_label = cez_graphs[year].df.class_label.apply(
        lambda x: class_value_to_label[x]
    )
```

```
Analysing year 2013
```

```
Identifying nodes: 100%|| 2923/2923 [00:05<00:00, 556.09it/s]
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 2003/2003 [00:10<00:00,␣
↪191.28it/s]
Step 2 of 2: Adding edges: 100%|| 2003/2003 [00:00<00:00, 69617.74it/s]
```

```
Graph successfully loaded with 2003 nodes and 5140 edges.
Analysing year 2014
```

```
Identifying nodes: 100%|| 2915/2915 [00:05<00:00, 541.60it/s]
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1999/1999 [00:11<00:00,␣
↪179.66it/s]
Step 2 of 2: Adding edges: 100%|| 1999/1999 [00:00<00:00, 60868.21it/s]
```

```
Graph successfully loaded with 1999 nodes and 5117 edges.
```

4. Creating `Timeline` and identifying nodes:

[8]:
```
cez_timeline = geotimeline.GeoGraphTimeline(cez_graphs)
# Perform node identification
cez_timeline.timestack()
# Classify node dynamics for the year 2014
cez_timeline.calculate_node_dynamics(2014);
```

```
Identifying nodes: 100%|| 2003/2003 [00:03<00:00, 514.92it/s]
```

5. Inspect in interactive viewer

[9]:
```
# Choose metrics to display:
metric_list = list(LANDSCAPE_METRICS_DICT.keys()) + list(COMPONENT_METRICS_DICT.
↪keys())
# Build up the viewer
viewer = geoviewer.GeoGraphViewer(small_screen=True, metric_list=metric_list)
viewer.add_layer(ipyleaflet.basemaps.Esri.WorldImagery)
viewer.add_graph(cez_timeline[2014], name="Chernobyl data", with_components=False)
viewer.enable_graph_controls()
```

```
Calculating component polygons...
```

```
Constructing graph: 100%|| 1/1 [00:00<00:00, 2499.59it/s]
```

```
Warning: very computationally expensive for graphs with more
             than ~100 components.
```

```
Constructing graph: 100%|| 1/1 [00:00<00:00, 3179.91it/s]
Calculating edge weights: 0it [00:00, ?it/s]
```

[10]:
```
# Display the interactive map
viewer
```

```
GeoGraphViewer(center=[51.389167, 30.099444], controls=(ZoomControl(options=['position
↪', 'zoom_in_text', 'zoom...
```

Note: an interactive viewer will show up here.

2.2.6 Creating Plots of Temporal Changes using Matplotlib

This tutorial shows how to create plots of dynamics and changes in GeoGraphs over time using Matplotlib.

1. Setup and Loading package

[2]:
```
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import numpy as np
import pandas as pd
import rioxarray as rxr
import geopandas as gpd
import pylandstats as pls
from geograph import GeoGraph
from geograph.geotimeline import GeoGraphTimeline
from geograph.constants import UTM35N
from geograph.demo.binder_constants import DATA_DIR, ROIS
from geograph.demo.plot_settings import ps_defaults, set_dim

ps_defaults(use_tex=True)

# Parse geotif landcover data
chernobyl_path = (
    lambda year: DATA_DIR / "chernobyl" / "esa_cci" / f"esa_cci_{year}_chernobyl.tif"
)

# Parse ROIS
rois = gpd.read_file(ROIS)
cez = rois[rois["name"] == "Chernobyl Exclusion Zone"]
ez = rois[rois.name.str.contains("Exclusion")]
```

2. Load Chernobyl Exclusion Zone data

[3]:
```
def clip_and_reproject(xrdata, clip_geometry=None, to_crs=UTM35N, x_res=300, y_
↪res=300):

    if clip_geometry is not None:
        clipped_data = xrdata.rio.clip(clip_geometry)
    else:
        clipped_data = xrdata

    if to_crs is not None:
        reprojected_data = clipped_data.rio.reproject(to_crs, resolution=(x_res, y_
↪res))
    else:
        reprojected_data = clipped_data

    return reprojected_data
```

[4]:
```
# Loading raster data
years = list(range(2000, 2015))
cez_rasters = {
    year: clip_and_reproject(
        rxr.open_rasterio(chernobyl_path(year)), clip_geometry=cez.geometry
    )
    for year in years
}
```

[5]:
```
## NOTE: For faster loading you can load the graphs from memory.
#        The demo path includes pre-loaded graphs for faster loading. Simply␣
↪uncomment.
# demo_path = DATA_DIR / "chernobyl" / "graphs"
# years = list(range(2000,2015))
# cez_graphs = {year: GeoGraph(chernobyl_path(yera))
#               for year in years}

# Polygonising raster and transforming into graph
cez_graphs = {}
for year, raster in cez_rasters.items():
    print(f"Analysing year {year}")
    # Load geograph from the raster data (construction takes ~5s)
    cez_graphs[year] = GeoGraph(
        data=raster.data.squeeze(),
        transform=raster.rio.transform(),
        mask=raster.data.squeeze() > 0,
        crs=UTM35N,
        connectivity=8,
    )
```

```
Analysing year 2000
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1924/1924 [00:01<00:00,␣
↪1425.51it/s]
Step 2 of 2: Adding edges: 100%|| 1924/1924 [00:00<00:00, 67463.43it/s]
Graph successfully loaded with 1924 nodes and 4912 edges.
Analysing year 2001
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1931/1931 [00:01<00:00,␣
↪1439.89it/s]
Step 2 of 2: Adding edges: 100%|| 1931/1931 [00:00<00:00, 54821.01it/s]
Graph successfully loaded with 1931 nodes and 4918 edges.
Analysing year 2002
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1929/1929 [00:01<00:00,␣
↪1413.43it/s]
Step 2 of 2: Adding edges: 100%|| 1929/1929 [00:00<00:00, 61787.40it/s]
Graph successfully loaded with 1929 nodes and 4897 edges.
Analysing year 2003
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1936/1936 [00:01<00:00,␣
↪1255.91it/s]
Step 2 of 2: Adding edges: 100%|| 1936/1936 [00:00<00:00, 64933.85it/s]
Graph successfully loaded with 1936 nodes and 4911 edges.
Analysing year 2004
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1953/1953 [00:01<00:00,␣
↪1082.23it/s]
Step 2 of 2: Adding edges: 100%|| 1953/1953 [00:00<00:00, 50851.88it/s]
Graph successfully loaded with 1953 nodes and 4953 edges.
Analysing year 2005
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1960/1960 [00:01<00:00,␣
↪1202.18it/s]
Step 2 of 2: Adding edges: 100%|| 1960/1960 [00:00<00:00, 58944.24it/s]
Graph successfully loaded with 1960 nodes and 4973 edges.
Analysing year 2006
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 2004/2004 [00:01<00:00,␣
↪1136.57it/s]
Step 2 of 2: Adding edges: 100%|| 2004/2004 [00:00<00:00, 61635.71it/s]
```

```
Graph successfully loaded with 2004 nodes and 5113 edges.
Analysing year 2007
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1996/1996 [00:01<00:00,␣
↪1206.78it/s]
Step 2 of 2: Adding edges: 100%|| 1996/1996 [00:00<00:00, 59821.01it/s]
```

```
Graph successfully loaded with 1996 nodes and 5141 edges.
Analysing year 2008
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1992/1992 [00:01<00:00,␣
↪1142.89it/s]
Step 2 of 2: Adding edges: 100%|| 1992/1992 [00:00<00:00, 39544.56it/s]
```

```
Graph successfully loaded with 1992 nodes and 5119 edges.
Analysing year 2009
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1994/1994 [00:01<00:00,␣
↪1236.88it/s]
Step 2 of 2: Adding edges: 100%|| 1994/1994 [00:00<00:00, 66390.22it/s]
```

```
Graph successfully loaded with 1994 nodes and 5108 edges.
Analysing year 2010
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1988/1988 [00:01<00:00,␣
↪1088.69it/s]
Step 2 of 2: Adding edges: 100%|| 1988/1988 [00:00<00:00, 50291.17it/s]
```

```
Graph successfully loaded with 1988 nodes and 5080 edges.
Analysing year 2011
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 2003/2003 [00:01<00:00,␣
↪1235.16it/s]
Step 2 of 2: Adding edges: 100%|| 2003/2003 [00:00<00:00, 61921.89it/s]
```

```
Graph successfully loaded with 2003 nodes and 5131 edges.
Analysing year 2012
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1998/1998 [00:01<00:00,␣
↪1203.33it/s]
Step 2 of 2: Adding edges: 100%|| 1998/1998 [00:00<00:00, 65800.49it/s]
```

```
Graph successfully loaded with 1998 nodes and 5119 edges.
Analysing year 2013
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 2003/2003 [00:01<00:00,␣
↪1127.01it/s]
Step 2 of 2: Adding edges: 100%|| 2003/2003 [00:00<00:00, 64449.54it/s]
```

```
Graph successfully loaded with 2003 nodes and 5140 edges.
Analysing year 2014
```

```
Step 1 of 2: Creating nodes and finding neighbours: 100%|| 1999/1999 [00:01<00:00,␣
↪1169.66it/s]
Step 2 of 2: Adding edges: 100%|| 1999/1999 [00:00<00:00, 65990.43it/s]
```

```
Graph successfully loaded with 1999 nodes and 5117 edges.
```

```
[ ]:
```

[6]:
```
cez_timeline = GeoGraphTimeline(cez_graphs)
```

[7]:
```
# Perform node identification between adjacent time slices (takes ~10s)
cez_timeline.timestack()
```

[7]:
```
[<src.binary_graph_operations.NodeMap at 0x7f5264602df0>,
 <src.binary_graph_operations.NodeMap at 0x7f52245212e0>,
 <src.binary_graph_operations.NodeMap at 0x7f5264670dc0>,
 <src.binary_graph_operations.NodeMap at 0x7f51f9c24400>,
 <src.binary_graph_operations.NodeMap at 0x7f5264670fa0>,
 <src.binary_graph_operations.NodeMap at 0x7f51f9c24280>,
 <src.binary_graph_operations.NodeMap at 0x7f5264664e80>,
 <src.binary_graph_operations.NodeMap at 0x7f52244ba670>,
 <src.binary_graph_operations.NodeMap at 0x7f51fa26f6a0>,
 <src.binary_graph_operations.NodeMap at 0x7f51fa27bd90>,
 <src.binary_graph_operations.NodeMap at 0x7f51fa017130>,
 <src.binary_graph_operations.NodeMap at 0x7f51f9e4f130>,
 <src.binary_graph_operations.NodeMap at 0x7f51fa255250>,
 <src.binary_graph_operations.NodeMap at 0x7f51fa255f40>]
```

3. Plots

Let us now visualise the ecosystem dynamics from 2013 to 2014

[8]:
```
# Identify node dynamics for the year 2014
cez_timeline.calculate_node_dynamics(2014)
```

[8]:
```
node_index
0       unchanged
1       unchanged
2       unchanged
3       unchanged
4       unchanged
          ...
1994    unchanged
1995    unchanged
1996    unchanged
1997    unchanged
1998    unchanged
Name: node_dynamic, Length: 1999, dtype: object
```

[9]:
```
cez_timeline[2014].df.node_dynamic.unique()
```

[9]:
```
array(['unchanged', 'split', 'birth', 'grew', 'shrank', 'complex',
       'merged'], dtype=object)
```

[10]:
```
graph = cez_timeline[2014]

plot_scale_factor = 1
dynamic_to_int = {
    "split": 0,
    "shrank": 1,
    "unchanged": 2,
    "complex": 3,
    "grew": 4,
    "merged": 5,
    "birth": 6,
}
```

(continues on next page)

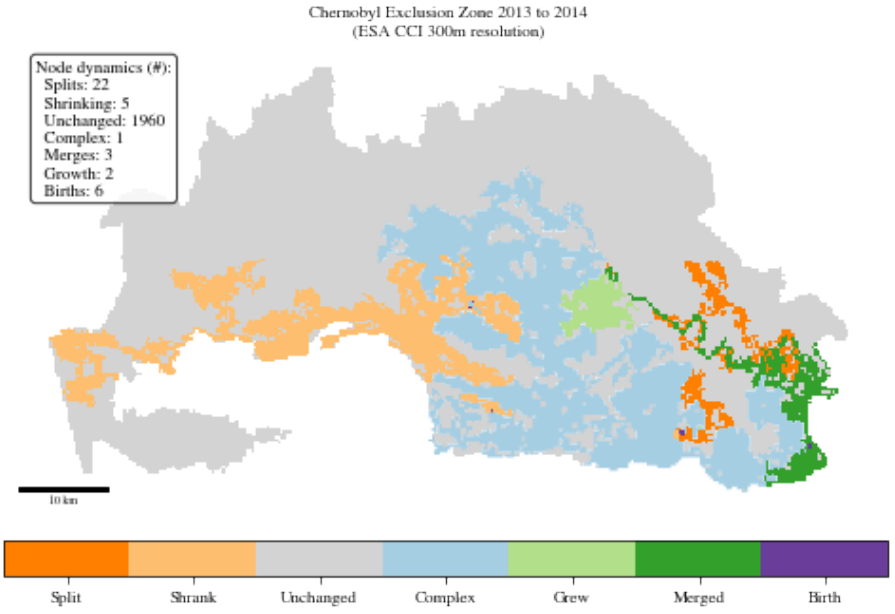
(continued from previous page)

```
colors = sns.color_palette("Paired").as_hex()
dynamic = lambda x: dynamic_to_int[x]
graph.df["dynamic_class"] = graph.df.node_dynamic.map(dynamic)

fig, ax = plt.subplots(1)
plt.title(
    "Chernobyl Exclusion Zone 2013 to 2014\n(ESA CCI 300m resolution)",
    fontsize=9 * plot_scale_factor,
)
set_dim(fig, fraction_of_line_width=plot_scale_factor)
vmin, vmax = 0, 7
cmap = mpl.colors.ListedColormap(
    [colors[7], colors[6], "lightgrey", colors[0], colors[2], colors[3], colors[9]]
)
graph.df.plot(column="dynamic_class", cmap=cmap, vmin=vmin, vmax=vmax, ax=ax)
ax.set_xticks([])
ax.set_yticks([])

inset_text = (
    "Node dynamics (#):\n"
    f"  Splits: {np.sum(graph.df['node_dynamic'] == 'split')}\n"
    f"  Shrinking: {np.sum(graph.df['node_dynamic'] == 'shrank')}\n"
    f"  Unchanged: {np.sum(graph.df['node_dynamic'] == 'unchanged')}\n"
    f"  Complex: {np.sum(graph.df['node_dynamic'] == 'complex')}\n"
    f"  Merges: {np.sum(graph.df['node_dynamic'] == 'merged')}\n"
    f"  Growth: {np.sum(graph.df['node_dynamic'] == 'grew')}\n"
    f"  Births: {np.sum(graph.df['node_dynamic'] == 'birth')}"
)

# these are matplotlib.patch.Patch properties
props = dict(boxstyle="round", facecolor="white", alpha=0.8)

# place a text box in upper left in axes coords
ax.text(
    0.03,
    0.97,
    inset_text,
    transform=ax.transAxes,
    fontsize=9 * plot_scale_factor,
    verticalalignment="top",
    bbox=props,
)

from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
import matplotlib.font_manager as fm

fontprops = fm.FontProperties(size=6 * plot_scale_factor)
scalebar = AnchoredSizeBar(
    ax.transData,
    1e4,
    "10 km",
    "lower left",
    pad=0,
    borderpad=0.8,
    color="black",
    frameon=False,
```

(continues on next page)

(continued from previous page)

```
    size_vertical=250,
    fontproperties=fontprops,
)

ax.add_artist(scalebar)

# add colorbar
fig = ax.get_figure()
cax = fig.add_axes(
    [0.12, 0.08, 0.78, 0.05]
)  # left-offset, # bottom offset # width, # height
sm = plt.cm.ScalarMappable(cmap=cmap, norm=plt.Normalize(vmin=vmin, vmax=vmax))
sm._A = []
cbar = fig.colorbar(sm, cax=cax, orientation="horizontal")
cbar.set_ticks(np.arange(vmin + 0.5, vmax + 1))
cbar.set_ticklabels(
    ["Split", "Shrank", "Unchanged", "Complex", "Grew", "Merged", "Birth"]
)
cbar.ax.tick_params(labelsize=9 * plot_scale_factor)
ax.spines["bottom"].set_visible(False)
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)


# plt.savefig("CEZ_node_dynamics.svg")
# plt.savefig("CEZ_node_dynamics.png")
```



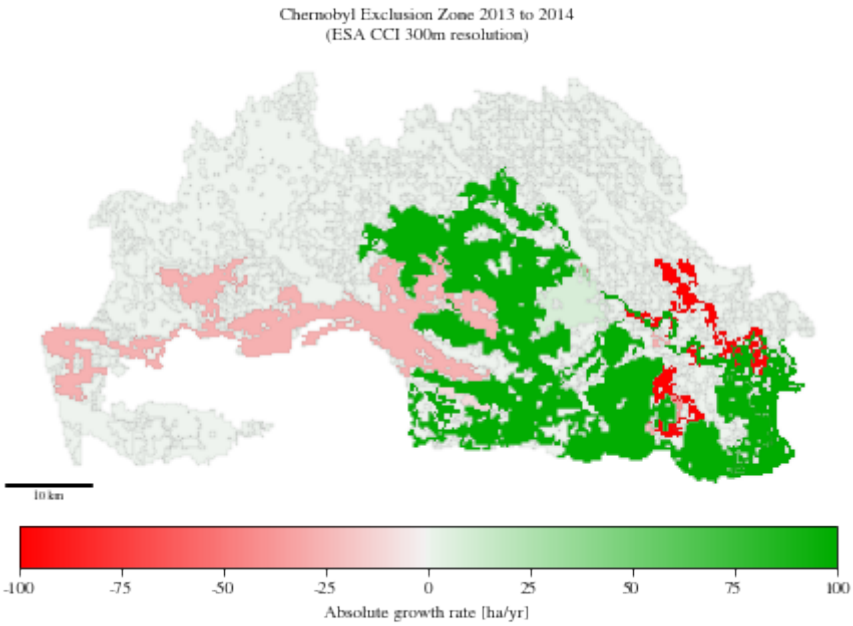
nbsphinx-code-borderwhite

[11]:
```
plot_scale_factor = 1
fig, ax = plt.subplots(1)
set_dim(fig, fraction_of_line_width=plot_scale_factor)

cmap = sns.diverging_palette(6, 120, s=360, l=55, as_cmap=True)
norm = mpl.colors.TwoSlopeNorm(vcenter=0, vmin=-10e5, vmax=10e5)

graph.df.plot(
```

(continues on next page)

(continued from previous page)

```
    "absolute_growth", ax=ax, cmap=cmap, norm=norm, edgecolor="grey", linewidth=0.1
)
ax.set_xticks([])
ax.set_yticks([])
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
import matplotlib.font_manager as fm

fontprops = fm.FontProperties(size=6 * plot_scale_factor)
scalebar = AnchoredSizeBar(
    ax.transData,
    1e4,
    "10 km",
    "lower left",
    pad=0,
    borderpad=0.3,
    color="black",
    frameon=False,
    size_vertical=250,
    fontproperties=fontprops,
)

ax.add_artist(scalebar)

cbar = fig.colorbar(
    mpl.cm.ScalarMappable(norm=norm, cmap=cmap),
    orientation="horizontal",
    label="Absolute growth rate [ha / yr]",
    # aspect=9,
    shrink=0.88,
    pad=0.04,
)
cbar.set_ticks(
    [-10e5, -7.5 * 1e5, -5e5, -2.5 * 1e5, 0, 2.5 * 1e5, 5e5, 7.5 * 1e5, 10e5]
)
cbar.set_ticklabels([-100, -75, -50, -25, 0, 25, 50, 75, 100])
cbar.ax.tick_params(labelsize=9 * plot_scale_factor)
cbar.set_label("Absolute growth rate [ha/yr]", fontsize=9 * plot_scale_factor)
plt.title(
    "Chernobyl Exclusion Zone 2013 to 2014\n(ESA CCI 300m resolution)",
    fontsize=9 * plot_scale_factor,
)
ax.spines["bottom"].set_visible(False)
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)
plt.savefig("CEZ_node_growth_rates.svg")
plt.savefig("CEZ_node_growth_rates.pdf")
plt.savefig("CEZ_node_growth_rates.png")
```



nbsphinx-code-borderwhite

[12]:
```
cez_timeline[2014].get_patch_metrics()
```

[12]:
```
            class_label      area  perimeter  perimeter_area_ratio  \
node_index
0                   100  270000.0     2400.0              0.008889
1                    70   90000.0     1200.0              0.013333
2                   130  180000.0     1800.0              0.010000
3                    30   90000.0     1200.0              0.013333
4                    30   90000.0     1200.0              0.013333
...                 ...       ...        ...                   ...
1994                 30  810000.0     6600.0              0.008148
1995                 60  180000.0     2400.0              0.013333
1996                100  180000.0     2400.0              0.013333
1997                160  180000.0     2400.0              0.013333
1998                 30  270000.0     3000.0              0.011111

            shape_index  fractal_dimension
node_index
0              1.154701           1.023003
1              1.000000           1.000000
2              1.060660           1.009734
3              1.000000           1.000000
4              1.000000           1.000000
...                 ...                ...
1994           1.833333           1.089106
1995           1.414214           1.057282
1996           1.414214           1.057282
1997           1.414214           1.057282
1998           1.443376           1.058689

[1999 rows x 6 columns]
```

[13]:
```
fig, ax = plt.subplots(1)
plot_scale_factor = 2
set_dim(fig, fraction_of_line_width=plot_scale_factor)
```

(continues on next page)

(continued from previous page)

```
cez_timeline[2013].df.loc[1629:1629].plot(ax=ax, color="red", alpha=0.6)
cez_timeline[2014].df.loc[1187:1187].plot(ax=ax, color="blue", alpha=0.6)
cez_timeline[2014].df.loc[1625:1625].plot(ax=ax, color="green", alpha=0.8)

inset_text = (
    f"Evergreen forest (70)\n\n"
    f"Total area: {graph.df.area.loc[1625]/1e4:.0f} ha\n"
    f"Perimeter: {graph.df.perimeter.loc[1625]/1e3:.0f} km\n"
    f"Fractal dimension: {graph.df.fractal_dimension.loc[1625]:.2f}\n"
    f"Shape index: {graph.df.shape_index.loc[1625]:.2f}"
)

# these are matplotlib.patch.Patch properties
props = dict(boxstyle="round", facecolor="white", alpha=0.8)

# place a text box in upper left in axes coords
ax.text(
    0.6,
    0.95,
    inset_text,
    transform=ax.transAxes,
    fontsize=8 * plot_scale_factor,
    verticalalignment="top",
    bbox=props,
)

ax.set_xticks([])
ax.set_yticks([])
ax.spines["bottom"].set_visible(False)
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)
plt.savefig("CEZ-nodediff-example.svg")
plt.savefig("CEZ-nodediff-example.pdf")
```

nbsphinx-code-borderwhite

2.3 GeoGraph API Reference

2.3.1 Subpackages

geograph.utils package

Submodules

geograph.utils.geopandas_utils module

Helper functions for operating with geopandas objects.

geograph.utils.geopandas_utils.**identify_dfs**(*df1*, *df2*, *mode*)

Idenitfy all nodes from *graph1* with nodes from *graph2* based on the given *mode*

Parameters

- **df1** (*GeoDataFrame*) – The dataframe whose node indicies will form the domain
- **df2** (*GeoDataFrame*) – The dataframe whose node indices will form the image (target)
- **mode** (*str*) – The mode to use for node identification. Must be one of *corner*, *edge* or *interior*. The different modes correspond to different rules for identification:
  - corner: Polygons of the same *class_label* which overlap, touch in their edges or corners will be identified with each other. (fastest)
  - edge: Polygons of the same *class_label* which overlap or touch in their edges will be identified with each other.
  - interior: Polygons of the same *class_label* which overlap will be identified with each other. Touching corners or edges are not counted.

Returns

A dictionary that represents the map from
elements of *df1* to *df2*.

Return type
mapping (Dict[int, np.ndarray])

geograph.utils.geopandas_utils.**identify_node**(*node*, *other_df*, *mode='corner'*)

Return list of all *loc* in *other_df* which identify with the given *node*.

Parameters

- **node** (*dict*) – The node for which to find nodes in *other_df* that can be identified with *node*.
- **other_df** (*GeoDataFrame*) – The GeoDataFrame object in which to search for identifications
- **mode** (*str, optional*) – Must be one of *corner*, *edge* or *interior*. Defaults to "corner". The different modes correspond to different rules for identification:
  - corner: Polygons of the same *class_label* which overlap, touch in their edges or corners will be identified with each other. (fastest)
  - edge: Polygons of the same *class_label* which overlap or touch in their edges will be identified with each other.
  - interior: Polygons of the same *class_label* which overlap will be identified with each other. Touching corners or edges are not counted.

Returns
List of node *loc* in *other_df* which identify with *node*.

Return type
np.ndarray

geograph.utils.geopandas_utils.**merge_diagonally_connected_polygons**(*df*)

Return a new dataframe with all geometries of *df* which touch at corners merged.

Merged geometries will be of type shapely.geometry.MultiPolygon

Parameters
df (*gpd.GeoDataFrame*) – The dataframe to analyse for geometries which touch at corners

Returns
The dataframe with patches that touch at corners merged

Return type
gpd.GeoDataFrame

geograph.utils.polygon_utils module

Helper functions for overlap computations with polygons in shapely.

geograph.utils.polygon_utils.**collapse_empty_polygon**(*polygon*)

Collapse *polygon* to an *EMPTY_POLYGON* if it is empty.

Parameters
polygon (*Polygon*) – The polygon to collapse if empty

Returns
Either the original, unchanges polygon or an empty polygon

Return type
Polygon

geograph.utils.polygon_utils.**connect_with_interior**(*polygon1*, *polygon2*)

Return True iff *polygon1* and *polygon2* overlap in interior, but not edge/corner.

Parameters

- **polygon1** (*Polygon*) – A shapely Polygon
- **polygon2** (*Polygon*) – The other shapely Polygon

Returns
True, iff *polygon1* and *polygon2* overlap in their interior.

Return type
bool

geograph.utils.polygon_utils.**connect_with_interior_bulk**(*polygon*, *polygon_array*)

Return boolean array with True iff polys overlap in interior, but not corner/edge.

Parameters

- **polygon** (*Polygon*) – A shapely Polygon
- **polygon_array** (*GeometryArray*) – The other shapely Polygons in a geopandas geometry array

Returns

Boolean array with value True, iff *polygon* and the polygon in *polygon_array* at the given location overlap in their interior.

Return type
List[bool]

geograph.utils.polygon_utils.**connect_with_interior_or_edge**(*polygon1*, *polygon2*)

Return True iff *polygon1* and *polygon2* overlap in interior/edge, but not corner.

Parameters

- **polygon1** (*Polygon*) – A shapely Polygon
- **polygon2** (*Polygon*) – The other shapely Polygon

Returns
True, iff *polygon1* and *polygon2* overlap in their interior/edge.

Return type
bool

geograph.utils.polygon_utils.**connect_with_interior_or_edge_bulk**(*polygon*, *polygon_array*)

Return boolean array with True iff polys overlap in interior/edge, but not corner.

Parameters

- **polygon** (*Polygon*) – A shapely Polygon
- **polygon_array** (*GeometryArray*) – The other shapely Polygons in a geopandas geometry array

Returns

Boolean array with value True, iff *polygon* and the polygon in
polygon_array at the given location overlap in their interior/edge.

Return type
List[bool]

geograph.utils.polygon_utils.**connect_with_interior_or_edge_or_corner**(*polygon1*, *polygon2*)

Return True iff *polygon1* and *polygon2* overlap in interior, edges or corners.

Parameters

- **polygon1** (*Polygon*) – A shapely Polygon
- **polygon2** (*Polygon*) – The other shapely Polygon

Returns
True, iff *polygon1* and *polygon2* intersect.

Return type
bool

geograph.utils.polygon_utils.**connect_with_interior_or_edge_or_corner_bulk**(*polygon*, *polygon_array*)

Return boolean array with True iff polygons overlap in interior, edges or corners.

Parameters

- **polygon** (*Polygon*) – A shapely Polygon
- **polygon_array** (*GeometryArray*) – The other shapely Polygons in a geopandas geometry array

Returns

Boolean array with value True, iff *polygon* and the polygon in
polygon_array at the given location intersect.

Return type
np.array

geograph.utils.polygon_utils.**de9im_match**(*pattern*, *target_pattern*)

Check a DE-9IM pattern *pattern* against a target DE-9IM pattern.

Note: To enable maximal speed, patterns are not parsed for correctness. For correct patterns consult https://en.wikipedia.org/wiki/DE-9IM.

Parameters

- **pattern** (*str*) – DE-9IM pattern to check as string
- **target_pattern** (*str*) – DE-9IM pattern against which to check as string

Returns

True, iff pattern matches with target_pattern

Return type

bool

geograph.utils.rasterio_utils module

A collection of utility functions for data loading with rasterio.

exception geograph.utils.rasterio_utils.**CoordinateSystemError**

Bases: Exception

Basic exception for coordinate system errors.

exception geograph.utils.rasterio_utils.**InvalidUseError**

Bases: Exception

Basic exception for invalid usage of functions.

geograph.utils.rasterio_utils.**get_thumbnail**(*data*, *band_idx=1*, *height=None*, *width=None*)

Calculate a thumbnail for a given band of a rasterio data.

Parameters

- **data** (*DatasetReader*) – rasterio data handle
- **band_idx** (*int, optional*) – The band index for which to calculate the
- **1.** (*thumbnail. Defaults to*) –
- **height** (*int, optional*) – The desired height of the thumbnail. If only the
- **set** (*height is*) –
- **datas** (*the width will be automatically determined from the*) –
- **100.** (*aspect ratio. Defaults to*) –
- **width** (*int, optional*) – The desired width of the thumbnail. Defaults to
- **None.** –

Returns

The 2D numpy array representing the thumbnail as calculated
from nearest neighbour resampling.

Return type

np.ndarray

geograph.utils.rasterio_utils.**polygonise**(*data_array*, *mask=None*, *transform=Affine(1.0, 0.0, 0.0, 0.0, 1.0, 0.0)*, *crs=None*, *connectivity=4*, *apply_buffer=True*)

Convert 2D numpy array containing raster data into polygons.

This implementation uses rasterio.features.shapes, which uses GDALpolygonize under the hood.

References: (1) https://rasterio.readthedocs.io/en/latest/api/rasterio.features.html (2) https://gdal.org/programs/gdal_polygonize.html

Parameters

- **data_array** (*np.ndarray*) – 2D numpy array with the raster data.
- **mask** (*np.ndarray, optional*) – Boolean mask that can be applied over
- **None.** (*resulting dataframe. Defaults to*) –
- **transform** (*affine.Affine, optional*) – Affine transformation to apply
- **transform.** (*when polygonising. Defaults to the identity*) –
- **crs** (*str, optional*) – Coordinate reference system to set on the
- **None.** –
- **connectivity** (*int, optional*) – Use 4 or 8 pixel connectivity for
- **4.** (*grouping pixels into features. Defaults to*) –
- **apply_buffer** (*bool, optional*) – Apply shapely buffer function to the
- **the** (*polygons after polygonising. This can fix issues with*) –
- **geometries.** (*polygonisation creating invalid*) –

Returns
GeoDataFrame containing polygon objects.

Return type
gpd.GeoDataFrame

geograph.utils.rasterio_utils.**read_from_lat_lon**(*data*, *band_idxs*, *lat*, *lon*, ***kwargs*)

Read in a tile of raster data form specified latitude and longitude values.

Note: This function only works if *data* is provided in the WGS geographical
coordinate system (Note: WGS84 = EPSG4326).

Parameters

- **data** (*DatasetReader*) – rasterio data handle
- **band_idxs** (*Union[int, Iterable[int]]*) – The band index or indices for which to read the information from the underlying rasterio *data*.
- **lat** (*Tuple[float]*) – A tuple containing (latitude_min, latitude_max). Latitudes must be in the range (-90, 90).
- **lon** (*Tuple[float]*) – A tuple containing (longitude_min, longitude_max). Longitudes must be in the range (-180, 180).

Returns

A multidimensional numpy array containing the specified bands in
the given latitude, longitude bounds.

Return type
np.ndarray

Module contents

geograph.visualisation package

Submodules

geograph.visualisation.control_widgets module

Module with widgets to control GeoGraphViewer.

class geograph.visualisation.control_widgets.**BaseControlWidget**(*viewer*)

Bases: Box

Base class for control widgets.

__init__(*viewer*)

Base class for control widgets.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to control

class geograph.visualisation.control_widgets.**CheckboxVisibilityWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget to control visibility of graphs in GeoGraphViewer with checkboxes.

__init__(*viewer*)

Widget to control visibility of graphs in GeoGraphViewer with checkboxes.

Note: this is currently not used by the main GraphControlWidget.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to control

class geograph.visualisation.control_widgets.**GraphControlWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget with full set of controls for GeoGraphViewer.

__init__(*viewer*)

Widget with full set of controls for GeoGraphViewer.

This is the control widget added to GeoGraphViewer. It is directly added to the viewer and combines other widgets such as visbility control, metrics, settings and more.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to control

class geograph.visualisation.control_widgets.**HoverWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget for showing patch information on mouse hover in GeoGraphViewer.

__init__(*viewer*)

Widget for showing patch information on mouse hover in GeoGraphViewer.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to show patch info in.

class geograph.visualisation.control_widgets.**LayerButtonWidget**(*viewer*, *layer_type*, *layer_subtype*, *layer_name=None*, *link_to_current_state=True*, *layout=None*, ***kwargs*)

Bases: ToggleButton

Toggle button to change the visibility of GeoGraphViewer layer.

__init__(*viewer*, *layer_type*, *layer_subtype*, *layer_name=None*, *link_to_current_state=True*, *layout=None*, ***kwargs*)

Toggle button to change the visibility of GeoGraphViewer layer.

Parameters

- **viewer** (geoviewer.GeoGraphViewer) – GeoGraphViewer to control
- **layer_type** (*str*) – type of layer
- **layer_subtype** (*str*) – subtype of layer
- **layer_name** (*Optional[str], optional*) – name of layer. Defaults to None. If None, the layer_name is automatically set to viewer.current_graph or viewer.current_map (depending on layer_type).
- **link_to_current_state** (*bool, optional*) – whether a traitlets link between the current state of the viewer and the button layer_name should be created. Defaults to True.
- **layout** (*Optional[widgets.Layout], optional*) – layout of the button. Defaults to None.

class geograph.visualisation.control_widgets.**MetricsWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget to show graph metrics in GeoGraphViewer.

__init__(*viewer*)

Widget to show graph metrics in GeoGraphViewer.

This widget shows metrics for *viewer.current_graph*.

Parameters

viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to show metrics for

class geograph.visualisation.control_widgets.**RadioVisibilityWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget to control visibility of graphs in GeoGraphViewer with radio buttons.

__init__(*viewer*)

Widget to control visibility of graphs in GeoGraphViewer with radio buttons.

This widget controls the visibility of graph as well as current map layers of GeoGraphViewer. Further, it sets the current_graph attribute of GeoGraphViewer that controls its state and is used by other widgets.

Parameters

viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to control

assemble_widget()

Assemble all sub-widgets making up VisibilityWidget into layout.

Returns

final widget to be added to GeoGraphViewer

Return type

widgets.Widget

create_visibility_buttons()

Create buttons that toggle the visibility of current graph and map.

The visibility of the current graph (set in self.current_graph), its subparts (e.g. components, disconnected nodes, etc.) and the map (set in self.current_map) can be controlled with the returned buttons. Separate buttons for the polygons and the components of the graph are included in the returned box.

Returns
box with button widgets

Return type
widgets.Box

class geograph.visualisation.control_widgets.**SettingsWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget for settings in GeoGraphViewer.

__init__(*viewer*)

Widget for settings in GeoGraphViewer.

Enables setting node size and color, and zoom level of viewer.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to show settings for

class geograph.visualisation.control_widgets.**TimelineWidget**(*viewer*)

Bases: *BaseControlWidget*

Widget to interact with GeoGraphTimeline.

__init__(*viewer*)

Widget to interact with GeoGraphTimeline.

Note: not fully implemented yet, currently just placeholder widget.

Parameters
viewer (geoviewer.GeoGraphViewer) – GeoGraphViewer to control

geograph.visualisation.folium_utils module

Module with utility functions to plot graphs in folium.

geograph.visualisation.folium_utils.**add_cez_to_map**(*folium_map*, *exclusion_json_path=None*, *add_layer_control=False*)

Add polygons of the Chernobyl Exclusion Zone (CEZ) to a folium map.

Parameters

- **folium_map** (*folium.Map*) – [description]
- **exclusion_json_path** (*Optional[str], optional*) – path to the json file containing the CEZ polygons. Defaults to None.
- **add_layer_control** (*bool, optional*) – whether to add layer controls to map. Warning: only use this when you don't intend to add any additional data after calling this function to the map. May cause bugs otherwise. Defaults to False.

Returns
map with CEZ polygons added

Return type
folium.Map

geograph.visualisation.folium_utils.**add_graph_to_folium_map**(*folium_map=None*, *polygon_gdf=None*, *color_column='index'*, *graph=None*, *name='data'*, *folium_tile_list=None*, *location=(51.389167, 30.099444)*, *crs='EPSG:32635'*, *add_layer_control=False*)

Create a visualisation map of the given polygons and *graph* in folium.

The polygons in *polygon_gdf* and *graph* are displayed on a folum map. It is intended that the graph was build from *polygon_gdf*, but it is not required. If given *map*, it will be put on this existing folium map.

Parameters

- **folium_map** (*folium.Map, optional*) – map to add polygons and graph to. Defaults to None.
- **polygon_gdf** (*gpd.GeoDataFrame, optional*) – data containing polygon. Defaults to None.
- **color_column** (*str, optional*) – column in polygon_gdf that determines which color is given to each polygon. Can be categorical values. Defaults to "index".
- **graph** (*Optional[geograph.GeoGraph], optional*) – graph to be plotted. Defaults to None.
- **name** (*str, optional*) – prefix to all the folium layer names shown in layer control of map (if added). Defaults to "data".
- **folium_tile_list** (*Optional[List[str]], optional*) – list of folium.Map tiles to be add to the map. See folium.Map docs for options. Defaults to None.
- **location** (*Tuple[float, float], optional*) – starting location in WGS84 coordinates Defaults to CHERNOBYL_COORDS_WGS84.
- **crs** (*str, optional*) – coordinates reference system to be used. Defaults to UTM35N.
- **add_layer_control** (*bool, optional*) – whether to add layer controls to map. Warning: only use this when you don't intend to add any additional data after calling this function to the map. May cause bugs otherwise. Defaults to False.

Returns
map with polygons and graph displayed as described

Return type
folium.Map

geograph.visualisation.folium_utils.**get_style_function**(*color='#ff0000'*)

Return lambda function that returns a dict with the *color* given.

The returned lambda function can be used as a style function for folium.

Parameters
color (*str, optional*) – color to be used in dict. Defaults to "#ff0000".

Returns
style function

Return type
Callable[[], dict]

geograph.visualisation.folium_utils.**remove_choropleth_color_legend**(*choropleth_map*)

Remove color legend from Choropleth folium map.

Solution proposed by *nhpackard* in the following GitHub issue in the folium repo: https://github.com/python-visualization/folium/issues/956

Parameters
choropleth_map (*folium.features.Choropleth*) – a Choropleth map

Returns
the same map without color legend

Return type
folium.features.Choropleth

geograph.visualisation.geoviewer module

This module contains the GeoGraphViewer to visualise GeoGraphs

class geograph.visualisation.geoviewer.**FoliumGeoGraphViewer**

Bases: `object`

Class for viewing GeoGraph object without ipywidgets

__init__()

Class for viewing GeoGraph object without ipywidgets.

add_graph(*graph*)

Add graph to viewer.

The added graph is visualised in the viewer.

Parameters
graph (geograph.GeoGraph) – GeoGraph to be shown

Return type
None

add_layer_control()

Add layer control to the viewer.

Return type
None

class geograph.visualisation.geoviewer.**GeoGraphViewer**(*center=(51.389167, 30.099444)*, *zoom=7*, *layout=None*, *metric_list=None*, *small_screen=True*, *logging_level='WARNING'*, *max_log_len=20*, *layer_update_delay=0.0*, ***kwargs*)

Bases: `Map`

Class for interactively viewing a GeoGraph.

__init__(*center=(51.389167, 30.099444)*, *zoom=7*, *layout=None*, *metric_list=None*, *small_screen=True*, *logging_level='WARNING'*, *max_log_len=20*, *layer_update_delay=0.0*, ***kwargs*)

Class for interactively viewing a GeoGraph.

Parameters

- **center** (*List[int, int], optional*) – center of the map. Defaults to CHERNOBYL_COORDS_WGS84.
- **zoom** (*int, optional*) – initial zoom level. Defaults to 7.
- **layout** (*Union[widgets.Layout, None], optional*) – layout passed to ipyleaflet.Map. Defaults to None.
- **metric_list** (*List[str], optional*) – list of GeoGraph metrics to be shown. Defaults to None.

- **small_screen** (*bool, optional*) – whether to reduce the control widget height for better usability on smaller screens. Defaults to True.
- **logging_level** (*str, optional*) – python logging level. Defaults to "WARNING".
- **max_log_len** (*int, optional*) – how many log messages should be displayed in in log tab. Note that a long log may slow down the viewer. Defaults to 20.
- **layer_update_delay** (*float, optional*) – how long the viewer should wait before updating layer. Whilst waiting other layer update requests are caught. This reduces the amount of traffic between the client (your browser) and the python kernel. Experimental. Defaults to 0.0.

add_graph(*graph*, *name='Graph'*, *with_components=True*)

Add GeoGraph to viewer.

Parameters

- **graph** (geograph.GeoGraph) – graph to be added
- **name** (*str, optional*) – name shown in control panel. Defaults to "Graph".
- **with_components** (*bool, optional*) – Iff True the graph components are calculated. Warning, this can make the loading of the viewer slow. Defaults to True.

Return type
None

add_layer(*layer*, *name=None*)

Add a layer on the map.

Parameters

- **layer** (*Layer instance*) – the new layer to add
- **name** (*str*) – name for the layer. This shows up in viewer control widgets.

Return type
None

enable_graph_controls()

Add controls for graphs to GeoGraphViewer.

Return type
None

hide_all_layers()

Hide all layers in self.layer_dict.

Return type
None

layer_update()

Update *self.layer* tuple from *self.layer_dict*.

Return type
None

request_layer_update()

Request layer_update to be called.

After receiving the first request the viewer waits for a set time, and then executes its layer_update method. If new requests come in whilst this time is passing no further action is taken. This helps avoid calling layer_update for each button in control widgets separately, slowing down the viewer.

set_graph_style(*radius=10*, *node_color=None*)

Set the style of any graphs added to viewer.

Parameters

- **radius** (*float*) – radius of nodes in graph. Defaults to 10.
- **node_color** (*str*) – (CSS) color of graph node (e.g. "blue")

Return type
None

set_layer_visibility(*layer_type*, *layer_name*, *layer_subtype*, *active*)

Set visiblity for a specific layer

Set the visibility for layer in *layer_dict[layer_type][layer_name][layer_subtype]*.

Parameters

- **layer_type** (*str*) – type of layer (e.g. "maps","graphs")
- **layer_name** (*str*) – name of layer
- **layer_subtype** (*str*) – subtype of layer (e.g. "map","components")
- **active** (*bool*) – whether layer is activate (=visible)

Return type
None

geograph.visualisation.graph_utils module

This module contains utility function for generally plotting graphs.

geograph.visualisation.graph_utils.**create_node_edge_geometries**(*graph*, *crs='EPSG:4326'*)

Create node and edge geometries for the networkx graph G.

Returns node and edge geometries in two GeoDataFrames. The output can be used for plotting a graph.

Parameters

- **graph** (GeoGraph) – graph with nodes and edges
- **crs** (*str, optional*) – coordinate reference system of graph. Defaults to UTM35N.

Returns

dataframes of nodes and edges
respectively.

Return type
Tuple[gpd.GeoSeries, gpd.GeoSeries]

geograph.visualisation.graph_utils.**map_dynamic_to_int**(*df*)

geograph.visualisation.style module

Module providing constants that define style of graph visualisation.

geograph.visualisation.widget_utils module

Module with utils for logging, debugging and styling ipywidgets.

class geograph.visualisation.widget_utils.**OutputWidgetHandler**(**args*, *max_len=30*, ***kwargs*)

Bases: Handler

Custom logging handler sending logs to an output widget.

Copied with minor adaptations from https://ipywidgets.readthedocs.io/en/latest/examples/Output%20Widget.html

__init__(**args*, *max_len=30*, ***kwargs*)

Initializes the instance - basically setting the formatter to None and the filter list to empty.

clear_logs()

Clear the current logs

emit(*record*)

Overload of logging.Handler method

show_logs()

Show the logs

geograph.visualisation.widget_utils.**create_html_header**(*text*, *level=1*)

Create html header widget from text.

Return type
HTML

Module contents

2.3.2 Submodules

2.3.3 geograph.binary_graph_operations module

Contains tools for binary operations between GeoGraph objects.

class geograph.binary_graph_operations.**NodeMap**(*src_graph*, *trg_graph*, *mapping*)

Bases: object

Class to store node mappings between two graphs (the src_graph and trg_graph).

__init__(*src_graph*, *trg_graph*, *mapping*)

Class to store node mappings between two graphs (*trg_graph* and *src_graph*).

This class stores a dictionary of node one-to-many relationships of nodes from *src_graph* to *trg_graph*. It also provides support for convenient methods for inverting the mapping and bundles the mapping information with references to the *src_graph* and *trg_graph*

Parameters

- **src_graph** (GeoGraph) – Domain of the node map (keys in *mapping* correspond to indices from the *src_graph*).
- **trg_graph** (GeoGraph) – Image of the node map (values in *mapping* correspond to indices from the *trg_graph*)
- **mapping** (*Dict[int, List[int]], optional*) – A lookup table for the map which maps nodes form *src_graph* to *trg_graph*.

invert()

Compute the inverse NodeMap from *trg_graph* to *src_graph*.

Return type
NodeMap

property mapping: Dict[int, List[int]]

Look-up table connecting node indices from *src_graph* to *trg_graph*.

Return type
Dict[int, List[int]]

property src_graph: *GeoGraph*

Keys in the mapping dict correspond to node indices in the *src_graph*.

Return type
GeoGraph

property trg_graph: *GeoGraph*

Values in the mapping dict correspond to node indices in the *trg_graph*.

Return type
GeoGraph

geograph.binary_graph_operations.**graph_polygon_diff**(*node_map*)

Return the polygons that were added/removed going from *src_graph* to *trg_graph*.

Parameters
node_map (NodeMap) – The node map from *src_graph* to *trg_graph*

Returns

Added parts and removed parts as geopandas
GeoDataFrame objects with the same index and crs as the src graph.

Return type
Tuple[GeoDataFrame, GeoDataFrame]

geograph.binary_graph_operations.**identify_graphs**(*graph1*, *graph2*, *mode*)

Idenitfy all nodes from *graph1* with nodes from *graph2* based on the given *mode*.

Parameters

- **graph1** (GeoGraph) – The GeoGraph whose node indicies will form the domain
- **graph2** (GeoGraph) – The GeoGraph whose node indices will form the image (target)
- **mode** (str) – The mode to use for node identification. Must be one of *corner*, *edge* or *interior*. The different modes correspond to different rules for identification:
  - corner: Polygons of the same *class_label* which overlap, touch in their edges or corners will be identified with each other. (fastest)
  - edge: Polygons of the same *class_label* which overlap or touch in their edges will be identified with each other.
  - interior: Polygons of the same *class_label* which overlap will be identified with each other. Touching corners or edges are not counted.

Returns
A NodeMap containing the map from *graph1* to *graph2*.

Return type
NodeMap

geograph.binary_graph_operations.**identify_node**(*node*, *other_graph*, *mode='corner'*)

Return list of all node ids in *other_graph* which identify with the given *node*.

Parameters

- **node** (*dict*) – The node for which to find nodes in *other_graphs* that can be identified with *node*.
- **other_graph** (GeoGraph) – The GeoGraph object in which to search for identifications
- **mode** (*str, optional*) – Must be one of *corner*, *edge* or *interior*. Defaults to "corner". The different modes correspond to different rules for identification:
  - corner: Polygons of the same *class_label* which overlap, touch in their edges or corners will be identified with each other. (fastest)
  - edge: Polygons of the same *class_label* which overlap or touch in their edges will be identified with each other.
  - interior: Polygons of the same *class_label* which overlap will be identified with each other. Touching corners or edges are not counted.

Returns

List of node ids in *other_graph* which identify with *node*.

Return type

List[int]

geograph.binary_graph_operations.**node_polygon_diff**(*src_node_id*, *node_map*)

Return the (multi)polygon areas that were added/removed from the given node.

Parameters

- **src_node_id** (*int*) – The id of the node in *src_graph* to check.
- **node_map** (NodeMap) – The node map object between *src_graph* and *trg_graph*

Returns

Added part and removed part as shapely
BaseGeometry objects.

Return type

Tuple[BaseGeometry, BaseGeometry]

2.3.4 geograph.constants module

All project wide constants are saved in this module.

2.3.5 geograph.geograph module

Module for processing and analysis of the geospatial graph.

See https://networkx.org/documentation/stable/index.html for graph operations.

class geograph.geograph.**ComponentGeoGraph**(*components_list*, *df=None*, *add_distance_edges=False*)

Bases: *GeoGraph*

Class to represent the connected components of a GeoGraph.

__init__(*components_list*, *df=None*, *add_distance_edges=False*)

Class for the connected components of a GeoGraph.

This class can load a graph from only a list of components, in which case it will not have a dataframe and the *has_df* class attribute will be False, or it can load from both a list of components and a dataframe.

In the latter case, edges with the polygon distance can be added between each pair of nodes (where each node corresponds to a connected component in the original graph).

WARNING: using *add_distance_edges=True* is incredibly slow for anything other than the smallest habitats.

Parameters

- **components_list** (*List[set]*) – A list of sets, where each set contains the node indices making up a component from the original graph.
- **df** (*Optional[gpd.GeoDataFrame], optional*) – A GeoDataFrame in which each row contains a polygon that represents a component from a GeoGraph. This is optional, and if not passed the graph will be created with no edges. Defaults to None.
- **add_distance_edges** (*bool, optional*) – Boolean that determines whether to add edges between every pair of nodes, with the distance between their corresponding polygons as an edge attribute. Defaults to False.

class geograph.geograph.**GeoGraph**(*data*, *crs=None*, *graph_save_path=None*, *raster_save_path=None*, *columns_to_rename=None*, *tolerance=0.0*, ***kwargs*)

Bases: object

Class for the fragmentation graph.

__init__(*data*, *crs=None*, *graph_save_path=None*, *raster_save_path=None*, *columns_to_rename=None*, *tolerance=0.0*, ***kwargs*)

Class for the fragmentation graph.

This class can load a pickled networkx graph directly, or create the graph from

- a path to vector data (.shp, .gpkg)
- a path to raster data (.tif, .tiff, .geotif, .geotiff)
- a numpy array containing raster data
- a dataframe containing polygons.

Note that the final dataframe must contain a class label column named "class_label" and a "geometry column containing the polygon data - the *columns_to_rename* argument allows for renaming columns to ensure this.

Warning: loading and saving GeoGraphs uses pickle. Loading untrusted data using the pickle module is not secure as it can execute arbitrary code. Therefore, only load GeoGraphs that come from a trusted source. See the pickle documentation for more details: https://docs.python.org/3/library/pickle.html

Parameters

- **data** – Can be a path to a pickle file or compressed pickle file to load the graph from, a path to vector data in GPKG or Shapefile format, a path to raster data in GeoTiff format, a numpy array containing raster data, or a dataframe containing polygons.
- **crs** (*str*) – Coordinate reference system to set on the resulting dataframe. Warning: whatever units of distance the CRS uses will be the units of distance for all polygon calculations, including for the *tolerance* argument. Using a lat-long CRS can therefore result in incoherent output.
- **graph_save_path** (*str or pathlib.Path, optional*) – A path to a pickle file to save the graph to, can be *.gz* or *.bz2*. Defaults to None, which will not save the graph.
- **raster_save_path** (*str or pathlib.Path, optional*) – A path to a file to save the polygonised raster data in. A path to a GPKG file is recommended, but Shapefiles also work. Defaults to None, which will not save the polygonised data.
- **columns_to_rename** (*Dict[str, str], optional*) – A dictionary mapping column names in the loaded dataframe with the new names of these columns. Use this

to ensure that the dataframe has "class_label" and "geometry" columns. Defaults to None.

- **tolerance** (*float, optional*) – Adds edges between neighbours that are at most *tolerance* units apart. Defaults to 0.
- ****mask** (*np.ndarray, optional*) – Boolean mask that can be applied over the polygonisation. Defaults to None.
- ****transform** (*affine.Affine, optional*) – Affine transformation to apply when polygonising. Defaults to the identity transform.
- ****connectivity** (*int, optional*) – Use 4 or 8 pixel connectivity for grouping pixels into features. Defaults to 4.
- ****apply_buffer** (*bool, optional*) – Apply shapely buffer function to the polygons after polygonising. This can fix issues with the polygonisation creating invalid geometries.

add_habitat(*name*, *valid_classes*, *barrier_classes=None*, *max_travel_distance=0.0*, *add_distance=False*, *add_component_edges=False*)

Create HabitatGeoGraph object and store it in habitats dictionary.

Creates a habitat subgraph of the main GeoGraph that only contains edges between nodes in *valid_classes* as long as they are less than *max_travel_distance* apart. All nodes which are not in *valid_classes* are not in the resulting habitat graph. This graph is then stored as its own HabitatGeoGraph object with all meta information.

The optional argument *barrier_classes* allows for a list of class labels which block the path between two nodes in *valid_classes*. Warning: The current pathfinding code is experimental and will only remove an edge between two classes within the max travel distance if the barrier class node in the middle *completely* blocks the path, which is rare. A full version of the pathfinding code is currently under development and will become available in a later release.

Warning: In a large dataset, passing values to *barrier_classes* will often make this function significantly slower, up to an order of magnitude.

Parameters

- **name** (*str*) – The name of the habitat.
- **valid_classes** (*List*) – A list of class labels which make up the habitat.
- **barrier_classes** (*List*) – A list of class labels which are barrier classes. The program will check if there are any nodes with a barrier class label completely blocking the path between two nodes less than *max_travel_distance* apart, and if so the edge will not be added. Note that this is only applicable in rare cases - in many instances the path will not be completely blocked and therefore the result will be the same as if there were no barrier classes. Defaults to None.
- **max_travel_distance** (*float*) – The maximum distance the animal(s) in the habitat can travel through non-habitat areas. The habitat graph will contain edges between any two nodes that have a class label in *valid_classes*, as long as they are less than *max_travel_distance* units apart. Defaults to 0, which will only create edges between directly neighbouring areas.
- **add_distance** (*bool, optional*) – Whether or not to add the distance between polygons as an edge attribute in the habitat graph. Defaults to False.
- **add_component_edges** (*bool, optional*) – Whether to add edges between nodes in the ComponentGeoGraph (which is automatically created as an attribute of the resulting HabitatGeoGraph) with edge weights that are the distance between neighbouring components. Can be computationally expensive. Defaults to False.

Raises

ValueError – If max_travel_distance < 0.

Return type
None

apply_to_habitats(*func*, ***kwargs*)

Apply a function to all habitats in this GeoGraph.

The function must be a method of GeoGraph or HabitatGeoGraph. **kwargs are applied to the function - all passed arguments must be specified with the keyword.

Parameters
func (*Callable*) – A function that is a method of GeoGraph or HabitatGeoGraph. This must not be a method of an instance of GeoGraph; it can only be an actual method definition, such as *GeoGraph.merge_nodes*.

Raises
ValueError – If *func* is not a method of GeoGraph or HabitatGeoGraph.

Returns
A list of the returned results.

Return type
List[Any]

property bounds

Return bounds of entire graph.

property class_label

Return class label of nodes directly from underlying numpy array.

Note: Uses *iloc* type indexing.

property classes: ndarray

Return a list of the sorted, unique class labels in the graph.

Return type
ndarray

property crs

Return crs of dataframe.

property geometry

Return geometry of nodes from underlying numpy array.

Note: Uses *iloc* type indexing.

get_class_metrics(*names=None*, *classes=None*, ***metric_kwargs*)

Return class-level metrics for the landcover classes in the given GeoGraph.

If arguments are omitted, all class level metrics are calculated.

Parameters

- **names** (*Optional[Union[str, Sequence[str]]], optional*) – Names of the metrics to calculate. If None, then all available class metrics are calculated for the given classses. Defaults to None.
- **classes** (*Optional[Union[str, Sequence[int]]], optional*) – Class labels of the classes to calculate. If None, then the given metrics are calculated for all classes. Defaults to None.
- ****metric_kwargs** – Any kwargs that should be passed on to the metrics.

Returns
A dataframe containing the metrics for the selected classes

Return type
pd.DataFrame

`get_graph_components`(*calc_polygons=True*, *add_distance_edges=False*)

Return a GeoGraph with the connected components of this graph.

This method determines the individual disconnected graph components that make up the graph of the GeoGraph object. It returns a ComponentGeoGraph object such that each row of the GeoDataFrame and each node in the networkx.Graph correspond to a connected component in the main graph, and the polygons in the dataframe are the union of all individual polygons making up a component in the main graph.

Warning: this method is very slow if *calc_polygons=True* and the graph consists mostly of one big component, since taking the union is expensive.

This method allows for the UI to visualise components and output their number as a metric.

More info on the definition of graph components can be found here: https://en.wikipedia.org/wiki/Component_(graph_theory)

Parameters

- **calc_polygons** (*bool, optional*) – This determines whether to calculate the polygons which are the union of all the polygons that make up each component, and load a *ComponentGeoGraph* with a corresponding dataframe containing these components. This can be time consuming if there is a very large component. Defaults to True.
- **add_distance_edges** (*bool, optional*) – This determines whether to add edges between every pair of nodes, with the distance between their corresponding polygons as an edge attribute. Defaults to False.

Returns

A ComponentGeoGraph containing the resulting
GeoDataFrame (if *calc_polygons=True*) and the list of graph components.

Return type
ComponentGeoGraph

`get_metric`(*name*, *class_value=None*, ***metric_kwargs*)

Calculate and save the metric with name *name* for the current GeoGraph.

Parameters

- **name** (*str*) – The name of a valid metric for a GeoGraph.
- **class_value** (*int*) – The landcover class label if a class level metric is desired. None if a landscape/component level metric is desired. Defaults to None.

Returns

The Metric object, containing the value as well as
other information about the metric.

Return type
metrics.Metric

`get_patch_metrics`()

Return patch-level metrics and append them to *self.df*.

Calculates “area”, “perimeter”, “perimeter_area_ratio”, “shape_index” and “fractal_dimension” for each patch

Returns
Dataframe containing the patch level metrics.

Return type
pd.DataFrame

identify_node(*node_id*, *other_graph*, *mode*)

Return all node ids in *other_graph* which identify with *node_id*.

Return type
List[int]

merge_classes(*class_list*, *new_name*)

Merge multiple classes together into one by renaming the class labels.

Warning: this can be very slow when merging classes with a lot of nodes.

Parameters

- **new_name** (*Union[str, int]*) – The new name for the combined class, either a string or an int.
- **class_list** (*List*) – The list of names of class labels to combine. Every name in the list must be in the GeoGraph.

Raises
ValueError – If *class_list* contains a class name not already in the GeoGraph.

Return type
None

merge_nodes(*node_list*, *class_label*, *final_index=None*)

Merge a list of nodes in the graph together into a single node.

This will create a node with a neighbour list and polygon which is the union of the nodes in *node_list*.

Parameters

- **node_list** (*List[int]*) – List of integer node indexes in the graph.
- **class_label** (*int or str*) – Class label for the resulting node.
- **final_index** (*int, optional*) – Index to assign to the resulting node. Defaults to None, in which case it becomes the highest valid index in the dataframe + 1.

Raises

- **ValueError** – If *final_index* is an existing node not in *node_list*,
- **or if node_list does not contain any existing nodes.** –

Return type
None

property rtree

Return Rtree object.

save_graph(*save_path*, *overwrite=False*, *pickle_protocol=4*)

Save graph with attributes and dataframe as pickle file. Can be compressed.

Parameters

- **save_path** (*Union[pathlib.Path, str]*) – Path to a pickle file. Can be compressed with gzip or bz2 by passing filenames ending in *gz* or *bz2*.
- **overwrite** (*bool, optional*) – If True, an existing file at *save_path* will be overwritten. Else throws an error. Defaults to False.
- **pickle_protocol** (*int, optional*) – Selects the pickle protocol that is used for python object serealisation. Supported protocols are explained here: https://docs.python.org/3/library/pickle.html#data-stream-format Defaults to pickle.DEFAULT_PROTOCOL (4 in python 3.8).

Raises
ValueError – If *save_path* is not a pickle, gz, or bz2 file.

Return type
None

class geograph.geograph.**HabitatGeoGraph**(*data*, *name=None*, *graph=None*, *valid_classes=None*, *barrier_classes=None*, *max_travel_distance=0*, *add_distance=False*, *add_component_edges=False*)

Bases: *GeoGraph*

Class to represent a habitat GeoGraph.

__init__(*data*, *name=None*, *graph=None*, *valid_classes=None*, *barrier_classes=None*, *max_travel_distance=0*, *add_distance=False*, *add_component_edges=False*)

Class to represent a habitat GeoGraph.

This class can load a habitat GeoGraph from a GeoDataFrame and networkx graph object, or alternatively load saved pickle or compressed pickle file with the graph, dataframe, and all metadata. Valid saved file formats are .pickle, .pkl, .gz, or .bz2.

Parameters

- **data** (Union[GeoDataFrame, str, PathLike]) – (GeoDataFrame or Path): Either a dataframe with the polygon data for the habitat graph nodes, or a path to a saved habitat. If it is a GeoDataFrame, then the other arguments in this init are mandatory (except for *add_distance* and *add_component_edges*)
- **name** (*str, optional*) – The name of the habitat.
- **graph** (*nx.Graph, optional*) – A networkx graph representing the habitat. Defaults to None.
- **valid_classes** (*List, optional*) – A list of class labels which make up the habitat.
- **barrier_classes** (*List, optional*) – A list of barrier class labels.
- **max_travel_distance** (*float, optional*) – The maximum distance the animal(s) in the habitat can travel through non-habitat areas. The habitat graph will contain edges between any two nodes that have a class label in *valid_classes*, as long as they are less than *max_travel_distance* units apart.
- **add_distance** (*bool, optional*) – Whether or not to add the distance between polygons has been added as an edge attribute in *graph*.
- **add_component_edges** (*bool, optional*) – Whether to add edges between nodes in the ComponentGeoGraph created automatically for this habitat with edge weights that are the distance between neighbouring components. Can be computationally expensive. Defaults to False.

Raises

- **ValueError** – If *data* is of an unknown type, or if *data* is a file
- **path of an invalid suffix.** –

save_habitat(*save_path*)

Save graph with attributes and dataframe as pickle file. Can be compressed.

Parameters

- **save_path** (*pathlib.Path*) – Path to a pickle file. Can be compressed
- **bz2.** (*with gzip or bz2 by passing filenames ending in gz or*) –

Raises
ValueError – If *save_path* is not a pickle, gz, or bz2 file.

Return type
None

2.3.6 geograph.geotimeline module

Module for analysing multiple GeoGraph objects.

class geograph.geotimeline.**GeoGraphTimeline**(*data*)

Bases: object

Timeline of multiple GeoGraphs.

__init__(*data*)

Creates a timeline of multiple GeoGraphs, for time-series and change-detection analyses.

The *data* must be a list of TimedGeoGraph objects or a dictionary where the keys correspond to the respective time-stamps of each GeoGraph. The GeoGraphs in *data* will be sorted from earliest to latest timestamp and added to the GeoGraphTimeline.

Landscape, Habitat-component, Class-value and patch level time-series analyses are supported.

Parameters

data (*Union[List[TimedGeoGraph], Dict[TimeStamp, GeoGraph]]*) – A list of TimedGeoGraphs or a dictionary where keys correspond to times and values to GeoGraph objects of the ecosystem at the specified time.

Raises

NotImplementedError – For any other *data* argument. (In the future we will add functionality to save timelines and load from disk.)

add_habitat(*name*, *valid_classes*, *barrier_classes=None*, *max_travel_distance=0.0*, *add_distance=False*, *add_component_edges=False*)

Create HabitatGeoGraph for each graph in the timeline.

Creates a habitat subgraph for each of the main GeoGraph objects in the timeline that only contains edges between nodes in *valid_classes* as long as they are less than *max_travel_distance* apart. All nodes which are not in *valid_classes* are not in the resulting habitat graph. This graph is then stored as its own HabitatGeoGraph object with all meta information.

Parameters

- **name** (*str*) – The name of the habitat.
- **valid_classes** (*List*) – A list of class labels which make up the habitat.
- **barrier_classes** (*List*) – Defaults to None.
- **max_travel_distance** (*as long as they are less than*) – The maximum distance the animal(s) in
- **graph** (*the habitat can travel through non-habitat areas. The habitat*) –
- **in** (*will contain edges between any two nodes that have a class label*) –
- **valid_classes** –
- **max_travel_distance** –
- **0** (*units apart. Defaults to*) –
- **between** (*which will only create edges*) –
- **areas.** (*directly neighbouring*) –
- **add_distance** (*bool, optional*) – Whether or not to add the distance
- **Defaults** (*between polygons as an edge attribute in the habitat graph.*) –
- **False.** (*computationally expensive. Defaults to*) –

- **add_component_edges** (*bool, optional*) – Whether to add edges between
- **an** (*nodes in the ComponentGeoGraph (which is automatically created as*) –
- **that** (*attribute of the resulting HabitatGeoGraph) with edge weights*) –
- **be** (*are the distance between neighbouring components. Can*) –
- **False.** –

Raises
ValueError – If max_travel_distance < 0.

Return type
None

calculate_node_dynamics(*time*)

Classify the dynamic behavior of each node of the graph at the given *time*.

The node dynamics reflects the type of change that a node has undergone between two adjacent timeslices of a GeoGraphTimeline. For each node at the selected *time* the classified node dynamics is one of:

birth: The child node has no ancestors in *nodemap* and newly appeared. *split*: The child node was created as a split off of an ancestral node. *unchanged*: The child node has one ancestor with the same characteristics. *grew*: The child node has one ancestor and has increased in area. *shrank*: The child node has one ancestor and has decreased in area. *complex*: The child node has numerous ancestors and siblings *merged*: The child node has several ancestors and no siblings. It was

created from a merge of several ancestral nodes.

Parameters
time (*TimeStamp*) – The time in GeoGraphTimeline for which the node dynamics should be classified. Must be one of the times in *self.times*.

Raises
UserWarning – If an inexistent timeslice is accessed.

Returns

A pandas series with the node dynamics for each node
the graph at the given time (self[time]).

Return type
pd.Series

get_class_metrics(*names=None*, *class_values=None*)

Return the time-series of the selected class metrics for the given *classes*.

Parameters

- **names** (*Optional[Union[str, Iterable[str]]], optional*) – Names of the class-level metrics to calculate. If None, all available class metrics are calculated. Defaults to None.
- **class_values** (*Optional[Union[int, Iterable[int]]], optional*) – Class values for the classes for which the metrics should be calculated. If None, the metrics are calcluated for all available classes in the GeoGraphTimeline. Classes which do not exist a certain point in time will have *np.nan* values. Defaults to None.

Returns

A three dimensional data array containing the time-series
class level metrics for the selected classes with dimensions (time, class_label, metric).

Return type
xr.DataArray

`get_metric`(*name*, *class_value=None*)

Return the time-series for the given metric.

For class-level metrics pass a class_value argument. For landscape/component level metrics omit the class_value argument.

Parameters

- **name** (*str*) – Name of the metric to compute
- **class_value** (*Optional[int], optional*) – Provide a class value if you wish
- **calculating** (*to calculate a class-level metric. Leave as None for*) –
- **None.** (*landscape/habitat level metrics. Defaults to*) –

Returns

A DataArray containing the metric time series for the graphs
in the given GeoGraphTimeline

Return type
xr.DataArray

`get_patch_metrics`(*aggregator='mean'*)

Return aggregated patch distribution metrics for all classes.

Parameters

- **aggregator** (*Union[str, Callable], optional*) – Aggregation function to use
- **"mean".** (*on the patch-level statistics. Defaults to*) –

Returns

A three dimensional array containing the time-series of the
aggregated patch-level distributions for each class. Dimension are (time, class_label, metric)

Return type
xr.DataArray

property graphs: Dict[Union[int, datetime], *GeoGraph*]

Return sorted list of GeoGraphs in this GeoGraphTimeline

Return type
Dict[Union[int, datetime], *GeoGraph*]

`identify_graphs`(*time1*, *time2*, *use_cached=True*)

Identify the nodes between the graph at time *time1* and *time2* in the timeline

Parameters

- **time1** (*TimeStamp*) – timestamp index of the first graph (will be src_graph)
- **time2** (*TimeStamp*) – timestamp index of the second graph (will be trg_graph)
- **use_cached** (*bool, optional*) – Iff True, use cached NodeMaps from previous computations. Defaults to True.

Returns
The one-to-many node mapping from *self[time1]* to *self[time2]*

Return type
NodeMap

node_diff_cache(*time1*, *time2*)

node_map_cache(*time1*, *time2*)

Return cached NodeMap from the graph at *time1* to that at *time2*.

Parameters

- **time1** (*TimeStamp*) – Time stamp of the first graph (src_graph)
- **time2** (*TimeStamp*) – Time stamp of the second graph (trg_graph)

Raises

NotCachedError – If the combination (time1, time2) or its inverse (time2, time1) have not been cached yet.

Returns

The NodeMap to identify nodes from *self[time1]* with *self[time2]*

Return type

NodeMap

timediff(*use_cached=True*)

property times: List[Union[int, datetime]]

Return list of valid time stamps for this GeoGraphTimeline

Return type

List[Union[int, datetime]]

timestack(*use_cached=True*)

Performs node identification between adjacent time-slices in the graph.

Parameters

use_cached (*bool, optional*) – If True, reuses prior node-identification computations. Defaults to True.

Returns

An ordered list of the of the node maps between each two
adjacent time-slices in the GeoGraphTimeline

Return type

List[*NodeMap*]

exception geograph.geotimeline.**NotCachedError**

Bases: Exception

Basic exception for values which were not yet cached.

class geograph.geotimeline.**TimedGeoGraph**(*time*, ***geographargs*)

Bases: *GeoGraph*

Wrapper class for GeoGraphs with a time attribute

__init__(*time*, ***geographargs*)

Simple wrapper class for GeoGraphs with a time attribute.

Parameters

- **time** (*TimeStamp*) – The timestamp of a given Geograph. Must be an integer or a python datetime object.
- ****geographargs** – Any argument to the GeoGraph class

property time: Union[int, datetime]

Return the time attribute.

Return type

Union[int, datetime]

2.3.7 geograph.metrics module

Functions for calculating metrics from a GeoGraph.

class geograph.metrics.**Metric**(*value*, *name*, *description*, *variant*, *unit=None*)

Bases: object

Class to represent a metric for a GeoGraph, with associated metadata.

__init__(*value*, *name*, *description*, *variant*, *unit=None*)

description: str

name: str

unit: Optional[str] = None

value: Any

variant: Optional[str]

2.3.8 Module contents

2.4 About

This package was created by Herbie Bradley, Arduin Findeis, Katherine Green, Yilin Li, Simon Mathis, and Simon Thomas, graduate students in the AI4ER CDT at the University of Cambridge. It was initially created as part of the AI4ER Group Team Challenge 2021.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

U

V