Time Series Analisys on Geoespatial Data with Python

Author: João Otavio Nascimento Firigato

email: joaootavionf007@gmail.com

LinkedIn: https://www.linkedin.com/in/jo%C3%A3o-otavio-firigato-4876b3aa/

First instructions:

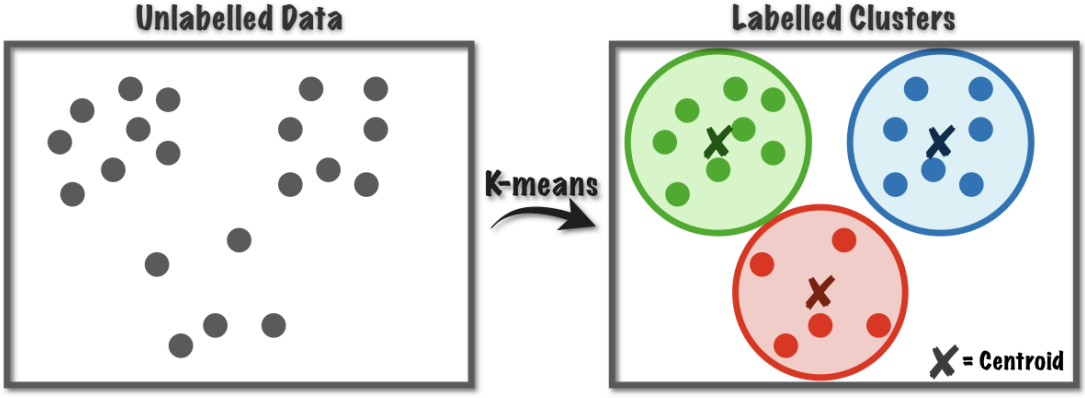
✅ Access the link to join our private WhatsApp community for students: https://chat.whatsapp.com/EPn27ZgR07lF3e1vnj8FiI

❗ It is important to access the Whatsapp Group to get the Colab Notebooks, as the PDF files are protected from text copying.

Chapter 11 - Time Series Clustering

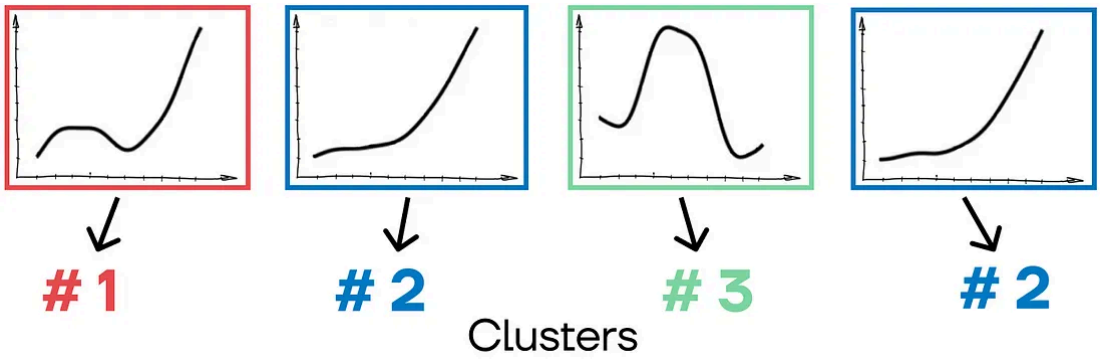
What is clustering?

Clustering is an unsupervised learning task that partitions a set of unlabeled data objects into homogeneous groups or clusters. Since we are in the unsupervised environment, we do not have advanced knowledge of the definitions or classifications of the groups. The goal of such a task is to form clusters in which individuals have maximum similarity to other individuals within the group and minimum similarity to individuals in other groups. This approach can be used for data mining, identifying structures in unlabeled datasets, data summarization, or as a preprocessing step in a more complex modeling system.



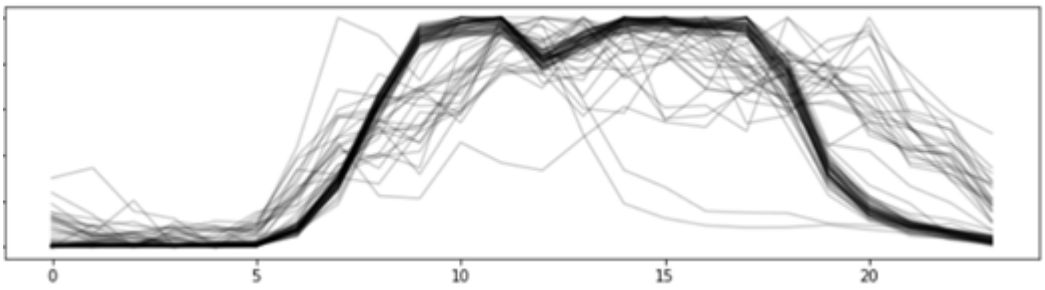
Time series clustering

Data for many applications are stored in time series format, for example: weather data, sales data, biomedical measurements like blood pressure and electrocardiogram, stock prices, biometric data, etc. Consequently, different works are found in a variety of domains, such as energy, finance, bioinformatics, or biology. Many relevant projects for analyzing time series have been carried out in various areas for different purposes, such as: subsequence matching, anomaly detection, motif discovery, indexing, visualization, segmentation, pattern identification, trend analysis, summarization, and forecasting.

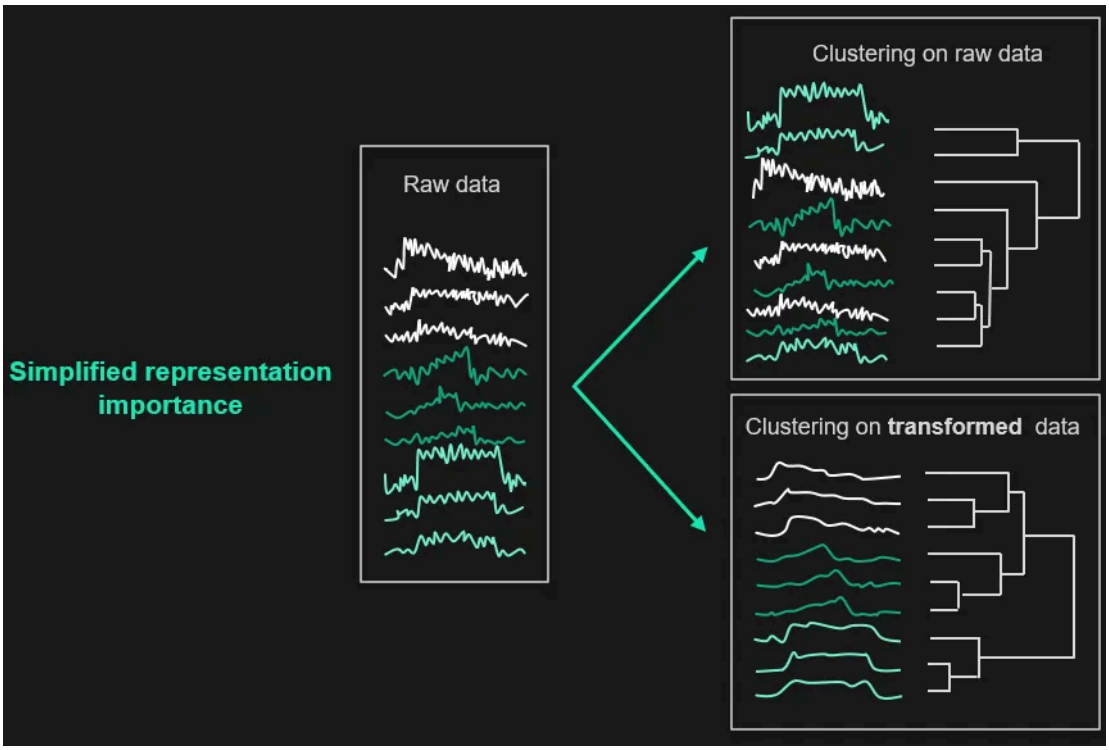


What are the main challenges?

Clustering time series in the context of large datasets is a difficult problem for two main reasons. First, time series data is often high-dimensional, which makes handling such data slow and difficult for many clustering algorithms. The second challenge concerns the similarity measures used to perform the clustering.



Methods for clustering time series



From raw series (grouping based on temporal proximity)

Classical clustering methods can be applied directly to the initial series.

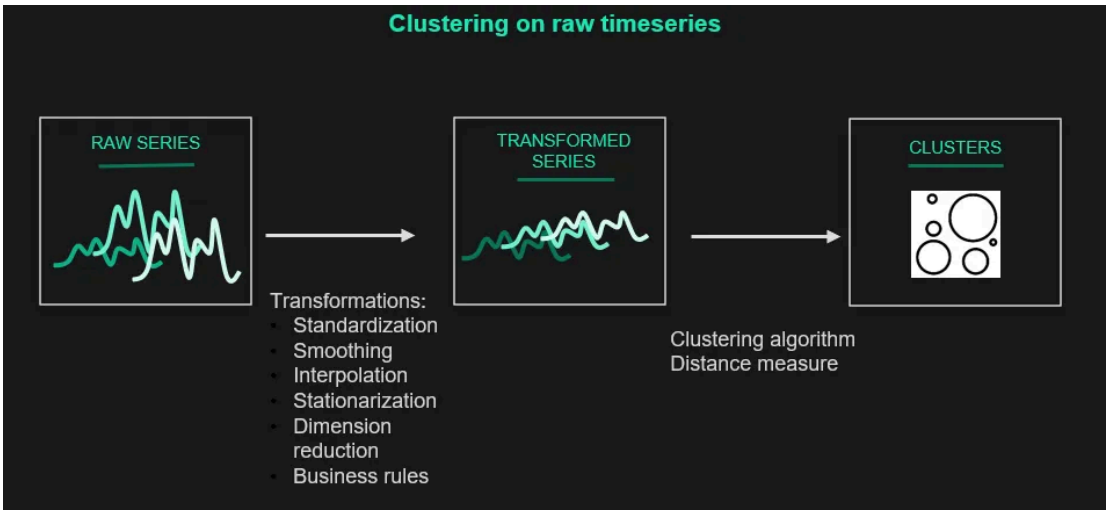
But be careful, in this case the choice of distance/similarity measure must be carefully considered before applying the algorithm. The selection involves determining what is meant by “similarity” between individual time series and can vary depending on the particularities of the data.

In practice, we rarely use the raw series directly. Instead, preliminary processing is performed to produce a simplified version of the series.

From a simplified representation of the series (representation-based grouping)

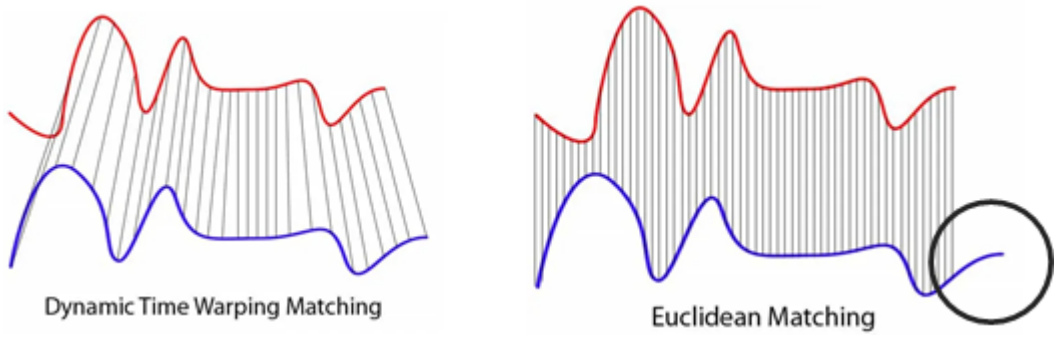
Simplified representation with rapid reduction of problem dimension

Raw data can be processed before clustering by applying transformations: standardization, smoothing, interpolation, stationarization, etc. These processes can remove noise and unwanted trends from the data and we can apply clustering algorithms to the resulting series.



Distance Measurements

Hundreds of distances have been proposed to classify data, of which Euclidean distance is the most popular. However, when it comes to clustering time series, using Euclidean distance, or any other Minkowski metric, on raw data can lead to counterintuitive results. In particular, this distance is very sensitive to scale effects and outliers or missing data points. Euclidean distance does not allow for temporal shifts to be taken into account. In fact, very similar signals can be shifted in time and would not be measured as “close” by the standard Euclidean distance. To solve this problem, there are adapted metrics, such as the DTW (Dynamic Time Warping) distance.



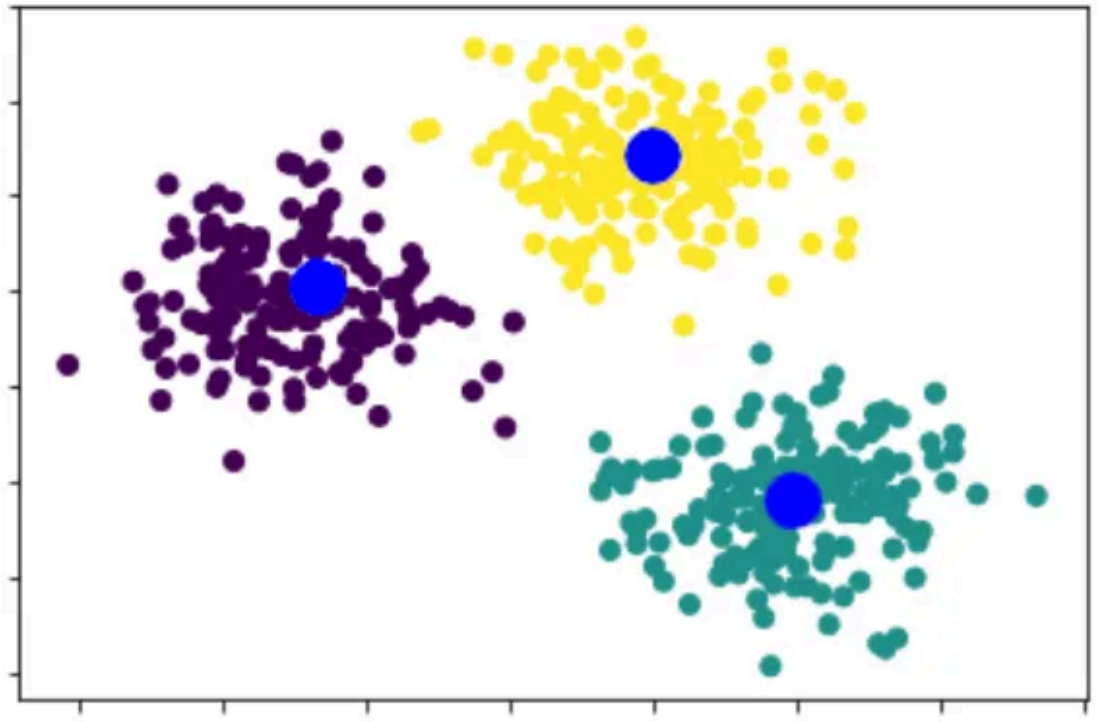
This distance takes into account non-linear elastic time and, consequently, accounts for relative temporal changes of two series by deforming the time axis. Intuitively, this means that this distance does not only compare points with corresponding timestamps. Thus, the Euclidean distance takes into account only simultaneous events, while the DTW distance takes into account out-of-phase events. Of course, it is not a magic measure that works in all cases. There is no single metric or clustering algorithm that can be considered superior to all others. It all depends on the context of the application, so it is up to you to experiment with different alternatives and choose the best one.

Depending on the measure selected, the resulting clusters may be "closer" in time or shape. These differences in clustering parameters can also be impacted by the data processing methods discussed above. Depending on the clustering objective and the length of the time series, the four data processing methods mentioned above can be

tested. Clustering can be applied to the raw time series, to a simplified representation of the series, to time series models, or to extracted features. Measures can then be applied impacting the temporal or geometric similarity of the clusters.

K-Means Algorithm

K-means clustering is a method that aims to group n inputs into k clusters in which each data point belongs to the cluster with the closest mean (cluster centroid). It can be visualized as Voronoi cells and is one of the most popular and most basic clustering algorithms.



Let's connect to the drive and GEE to get data for our example:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
import ee
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import geemap
import geopandas as gpd
```

In [ ]:
```
ee.Authenticate()
ee.Initialize(project='my-project-1527255156007')
```

We will use point files of four types of classes:

Download link:
https://drive.google.com/drive/folders/1Vn4w8WqHgHta7rwJ5gD_WRARo7b1SLAv?

usp=sharing

In [ ]:
```
path_agua = '/content/drive/MyDrive/Datasets_TS/Pontos_LST/Pontos_agua.shp'
path_cana = '/content/drive/MyDrive/Datasets_TS/Pontos_LST/Pontos_cana.shp'
path_cidade = '/content/drive/MyDrive/Datasets_TS/Pontos_LST/Pontos_cidade.shp'
path_reserva = '/content/drive/MyDrive/Datasets_TS/Pontos_LST/Pontos_reserva.shp
```

In [ ]:
```
gpd_agua = gpd.read_file(path_agua)
gpd_cana = gpd.read_file(path_cana)
gpd_cidade = gpd.read_file(path_cidade)
gpd_reserva = gpd.read_file(path_reserva)
```

In [ ]:
```
gpd_agua['id'] = 0
gpd_cana['id'] = 1
gpd_cidade['id'] = 2
gpd_reserva['id'] = 3
```

Let's join all the points into a single DataFrame:

In [ ]:
```
gdf_full = pd.concat([gpd_agua, gpd_cana, gpd_cidade, gpd_reserva])
```

We converted it to Feature Collection and presented it with Geemap:

In [ ]:
```
fc = geemap.geopandas_to_ee(gdf_full)
```

In [ ]:
```
Map = geemap.Map()
Map.centerObject(fc, 10)
Map.addLayer(fc, {}, "Points")
Map
```

Let's get monthly data from 2019 to 2021:

In [ ]:
```
months = ee.List.sequence(1,12)
years = ee.List.sequence(2019, 2021)
```

In [ ]:
```
landsat_collection = ee.ImageCollection('LANDSAT/LC08/C02/T1_L2').filterDate('20
```

In [ ]:
```
def calculate_ndvi(image):
  # Calculate NDVI
  ndvi = image.normalizedDifference(['SR_B5', 'SR_B4']).rename('NDVI')
  # Add NDVI band to the image
  return image.addBands(ndvi)

landsat_ndvi = landsat_collection.map(calculate_ndvi)
```

In [ ]:
```
def monthly(collection):
  img_coll = ee.ImageCollection([])
  for y in years.getInfo():
    for m in months.getInfo():
      filtered = collection.filter(ee.Filter.calendarRange(y, y, 'year')).filter
      filtered = filtered.median()
      img_coll = img_coll.merge(filtered.set('year', y).set('month', m).set('sys
  return img_coll
```

We apply our function and obtain the monthly NDVI collection from Landsat 8:

In [ ]:
```
Monthly_MD = monthly(landsat_ndvi)
```

We can check the bands for each image:

In [ ]:
```
image_list = Monthly_MD.toList(Monthly_MD.size())
for i in range(image_list.size().getInfo()):
  image = ee.Image(image_list.get(i))
  band_names = image.bandNames().getInfo()
  print(f"Image {i+1}: {band_names}")
```

```
Image 1: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 2: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 3: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 4: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 5: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 6: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 7: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 8: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 9: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_A
EROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_Q
A', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 10: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 11: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 12: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 13: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 14: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 15: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 16: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 17: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 18: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 19: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 20: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
```

```
Image 21: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 22: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 23: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 24: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 25: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 26: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 27: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 28: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 29: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 30: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 31: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 32: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 33: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 34: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 35: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
Image 36: ['SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7', 'SR_QA_
AEROSOL', 'ST_B10', 'ST_ATRAN', 'ST_CDIST', 'ST_DRAD', 'ST_EMIS', 'ST_EMSD', 'ST_
QA', 'ST_TRAD', 'ST_URAD', 'QA_PIXEL', 'QA_RADSAT', 'NDVI']
```

And check the date of each image:

In [ ]:
```python
def print_image_dates(image_collection):
  image_list = image_collection.toList(image_collection.size())
  for i in range(image_collection.size().getInfo()):
    image = ee.Image(image_list.get(i))
    date_ms = image.get('system:time_start').getInfo()
    date = ee.Date(date_ms).format('YYYY-MM-dd').getInfo()
    print(f"Image {i+1}: {date}")
```

```
print_image_dates(Monthly_MD)
```

```
Image 1: 2019-01-01
Image 2: 2019-02-01
Image 3: 2019-03-01
Image 4: 2019-04-01
Image 5: 2019-05-01
Image 6: 2019-06-01
Image 7: 2019-07-01
Image 8: 2019-08-01
Image 9: 2019-09-01
Image 10: 2019-10-01
Image 11: 2019-11-01
Image 12: 2019-12-01
Image 13: 2020-01-01
Image 14: 2020-02-01
Image 15: 2020-03-01
Image 16: 2020-04-01
Image 17: 2020-05-01
Image 18: 2020-06-01
Image 19: 2020-07-01
Image 20: 2020-08-01
Image 21: 2020-09-01
Image 22: 2020-10-01
Image 23: 2020-11-01
Image 24: 2020-12-01
Image 25: 2021-01-01
Image 26: 2021-02-01
Image 27: 2021-03-01
Image 28: 2021-04-01
Image 29: 2021-05-01
Image 30: 2021-06-01
Image 31: 2021-07-01
Image 32: 2021-08-01
Image 33: 2021-09-01
Image 34: 2021-10-01
Image 35: 2021-11-01
Image 36: 2021-12-01
```

Now we convert the series of images to a DataFrame::

In [ ]:
```
feature_list = fc.toList(fc.size())
dfs = []

for i in range(fc.size().getInfo()):
  feature = ee.Feature(feature_list.get(i))
  class_id = feature.get('id').getInfo()
  time_series = Monthly_MD.select('NDVI').getRegion(feature.geometry(), 30).getI
  df = pd.DataFrame(time_series[1:], columns=time_series[0])
  df = df[['longitude', 'latitude', 'time', 'NDVI']]
  df.rename(columns={'time':'date'}, inplace=True)

  # Convert the date from milliseconds to datetime objects.
  df['date'] = pd.to_datetime(df['date'], unit='ms')
  df = df.set_index('date')
  df['feature_id'] = i
  df['class_id'] = class_id
  dfs.append(df)
```

```
final_df = pd.concat(dfs)
final_df
```

Out[ ]:

date	longitude	latitude	NDVI	feature_id	class_id
2019-01-01	-50.657301	-20.926120	0.094833	0	0
2019-02-01	-50.657301	-20.926120	0.116266	0	0
2019-03-01	-50.657301	-20.926120	0.041275	0	0
2019-04-01	-50.657301	-20.926120	-0.003463	0	0
2019-05-01	-50.657301	-20.926120	0.042498	0	0
...	...	...	...	...	...
2021-08-01	-50.521746	-21.084044	0.169276	105	3
2021-09-01	-50.521746	-21.084044	0.296380	105	3
2021-10-01	-50.521746	-21.084044	0.311862	105	3
2021-11-01	-50.521746	-21.084044	0.378156	105	3
2021-12-01	-50.521746	-21.084044	0.402175	105	3

3816 rows × 5 columns

In [ ]:
```
final_df.drop(columns={'longitude','latitude'}, inplace=True)
```

In [ ]:
```
final_df['date'] = final_df.index
```

We can plot the time series of each point by their respective classes:

In [ ]:
```
import plotly.express as px


fig = px.line(final_df, x='date', y='NDVI', color='feature_id', facet_col='class
              labels={'date': 'Date', 'NDVI':'NDVI', 'feature_id':'Feature ID',
              title='Time Series of NDVI by Feature ID and Class ID')
fig.show()
```

We pivot our Dataframe to get the columns as Dates:

In [ ]:
```
pivoted_df = final_df.pivot(index='feature_id', columns='date', values='NDVI')
pivoted_df
```

Out[ ]:

date	2019-01-01	2019-02-01	2019-03-01	2019-04-01	2019-05-01	2019-06-01	2019-07-01	201
feature_id								
0	0.094833	0.116266	0.041275	-0.003463	0.042498	-0.019456	-0.011366	-0.0
1	0.061652	0.132942	0.008162	-0.005530	0.063989	-0.009637	-0.006807	-0.0
2	0.094250	0.063221	-0.009423	-0.009157	0.107729	-0.015520	-0.013755	-0.0
3	0.109927	0.003998	-0.010714	-0.009956	0.096144	-0.013231	-0.009510	-0.0
4	0.086098	0.007734	0.005219	-0.010751	0.096271	-0.015009	-0.013761	-0.0
...	...	...	...	...	...	...	...	
101	0.151086	0.272270	0.431394	0.375502	0.379856	0.336836	0.302526	0.1
102	0.159619	0.248704	0.326311	0.318180	0.330156	0.321391	0.303998	0.1
103	0.146223	0.278045	0.417649	0.363990	0.370138	0.306579	0.275267	0.1
104	0.217346	0.402631	0.360071	0.427471	0.334912	0.348880	0.328537	0.3
105	0.217331	0.336104	0.353571	0.416882	0.349251	0.380856	0.363016	0.3

106 rows × 36 columns

Before we apply KMeans, let's standardize the data:

In [ ]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
```

In [ ]:
```
scaler = StandardScaler()
time_series_data_scaled = scaler.fit_transform(pivoted_df)
```

We apply KMeans and obtain the labels as a result:

In [ ]:
```
kmeans = KMeans(n_clusters=4, random_state=0)
labels = kmeans.fit_predict(time_series_data_scaled)

print(labels)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 0 0 2 2 0 0 0 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 1 1 1 1 1 1 2 1 1
 2 1 1 2 1 1 1 2 1 1 1 2 2 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3]
```

K-Means with DTW

To use K-Means specifically for time series, let's install the tslean library:

In [ ]:
```
!pip install tslearn
```

```
Collecting tslearn
  Downloading tslearn-0.6.3-py3-none-any.whl.metadata (14 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from tslearn) (2.0.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages
(from tslearn) (1.14.1)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-pac
kages (from tslearn) (1.6.1)
Requirement already satisfied: numba in /usr/local/lib/python3.11/dist-packages
(from tslearn) (0.60.0)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages
(from tslearn) (1.4.2)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/pytho
n3.11/dist-packages (from numba->tslearn) (0.43.0)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/
dist-packages (from scikit-learn->tslearn) (3.6.0)
Downloading tslearn-0.6.3-py3-none-any.whl (374 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 374.4/374.4 kB 6.2 MB/s eta 0:00:00
Installing collected packages: tslearn
Successfully installed tslearn-0.6.3
```

We import the modules and functions used:

In [ ]:
```
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
from tslearn.clustering import TimeSeriesKMeans
from tslearn.utils import to_time_series_dataset
```

We convert our dataframe to the format that tslearn needs:

In [ ]:
```
time_series_dataset = to_time_series_dataset(pivoted_df)
```

In [ ]:
```
time_series_dataset.shape
```

Out[ ]:
```
(106, 36, 1)
```

We apply standardization:

In [ ]:
```
scaler = TimeSeriesScalerMeanVariance()
time_series_dataset_scaled = scaler.fit_transform(time_series_dataset)
```

And we can apply TimeSeriesKMeans using the DTW distance:

In [ ]:
```
model = TimeSeriesKMeans(n_clusters=4, metric="dtw", random_state=0)
labels = model.fit_predict(time_series_dataset_scaled)

# Display cluster labels
print(labels)
```

```
[3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2
 2 2 0 0 2 2 0 0 0 2 0 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 1 1 1 1 1 2 1 2 1 1
 2 1 1 2 1 1 1 0 2 1 2 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0]
```

In [ ]:
```
gdf_full['classe'] = labels
```

In [ ]:
```
gdf_full
```

Out[ ]:

	id	geometry	classe
0	0	POINT (-50.65727 -20.92599)	3
1	0	POINT (-50.61445 -20.94601)	3
2	0	POINT (-50.58219 -20.96881)	3
3	0	POINT (-50.55605 -20.98939)	3
4	0	POINT (-50.54549 -21.00385)	3
...	...	...	...
13	3	POINT (-50.56653 -21.17921)	2
14	3	POINT (-50.56923 -21.1956)	0
15	3	POINT (-50.56276 -21.17831)	0
16	3	POINT (-50.52583 -21.08301)	0
17	3	POINT (-50.52178 -21.08402)	0

106 rows × 3 columns

Using Folium, we present the classified points:

In [ ]:
```
import folium
basemaps = {
    'Google Maps': folium.TileLayer(
        tiles = 'https://mt1.google.com/vt/lyrs=m&x={x}&y={y}&z={z}',
        attr = 'Google',
        name = 'Google Maps',
        overlay = True,
        control = True
    ),
    'Google Satellite': folium.TileLayer(
        tiles = 'https://mt1.google.com/vt/lyrs=s&x={x}&y={y}&z={z}',
        attr = 'Google',
        name = 'Google Satellite',
        overlay = True,
        control = True
    ),
    'Google Terrain': folium.TileLayer(
        tiles = 'https://mt1.google.com/vt/lyrs=p&x={x}&y={y}&z={z}',
        attr = 'Google',
        name = 'Google Terrain',
        overlay = True,
        control = True
    ),
    'Google Satellite Hybrid': folium.TileLayer(
        tiles = 'https://mt1.google.com/vt/lyrs=y&x={x}&y={y}&z={z}',
        attr = 'Google',
        name = 'Google Satellite',
        overlay = True,
        control = True
    ),
    'Esri Satellite': folium.TileLayer(
        tiles = 'https://server.arcgisonline.com/ArcGIS/rest/services/World_Imag
```

```
        attr = 'Esri',
        name = 'Esri Satellite',
        overlay = True,
        control = True
    )
}
```

In [ ]:

```
resultmap = folium.Map(location=[gdf_full.geometry.y.values[0],gdf_full.geometry
basemaps['Google Satellite Hybrid'].add_to(resultmap)

latitudes = list(gdf_full.geometry.y.values)
longitudes = list(gdf_full.geometry.x.values)
labels = list(gdf_full['classe'].values)
for lat, lng, label in zip(latitudes, longitudes, labels):
  if label == 0:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='red')
     ).add_to(resultmap)
  elif label == 1:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='blue')
     ).add_to(resultmap)
  elif label == 2:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='black')
     ).add_to(resultmap)
  else:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='green')
     ).add_to(resultmap)


vis_params = {'min': 0, 'max': 1}# Add the elevation model to the map object.
resultmap.add_child(folium.LayerControl())

# Display the map.
display(resultmap)
```

DBSCAN

The DBSCAN model, which stands for “Density-Based Spatial Clustering of Applications with Noise”, has a density-based clustering approach. The density of points in a given region is responsible for the formation of clusters. If a given point does not meet density criteria or distance threshold criteria, it cannot be classified into a cluster.

The two parameters required to apply the DBSCAN model are:

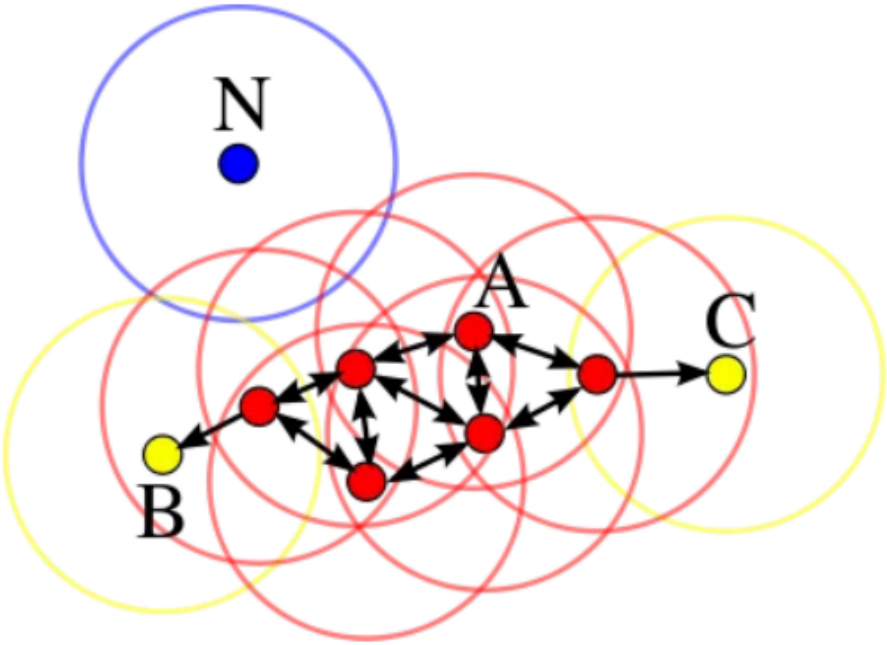
- Eps: Maximum radius within which two points can be considered to be in the same cluster.

- MinPts: Minimum number of points in a region required to ensure the desired density.

The choice of parameters is crucial for the algorithm to work properly. Choosing an Eps that is too small will lead to a lot of noise, since it will not be possible to classify these more distant points. However, choosing a high value may lead to the classification of few very generalized clusters. Techniques such as k-distance graphs can help define the best Eps value. Now, when it comes to MinPts, it is recommended to decide the value based on the size of the dataset. Higher values for this parameter are effective for datasets with a lot of noise.

In the parameters section, the presence of noise was already mentioned. This is because the classification of a point in the dataset is crucial for the application of the algorithm. There are three types of points that we can find:

- Cores: A point is considered a core when it has MinPts points or more within a radius of Eps of distance.
- Edges: A point is considered a border when it does not have MinPts within a radius of Eps of distance, but is inserted within a radius of Eps of another point in which it is a core.
- Noise (or Outliers): A point is considered noise when it is not present in the radius Eps of a core and does not have MinPts within its radius Eps.



In [ ]:
```
pivoted_df
```

Out[ ]:

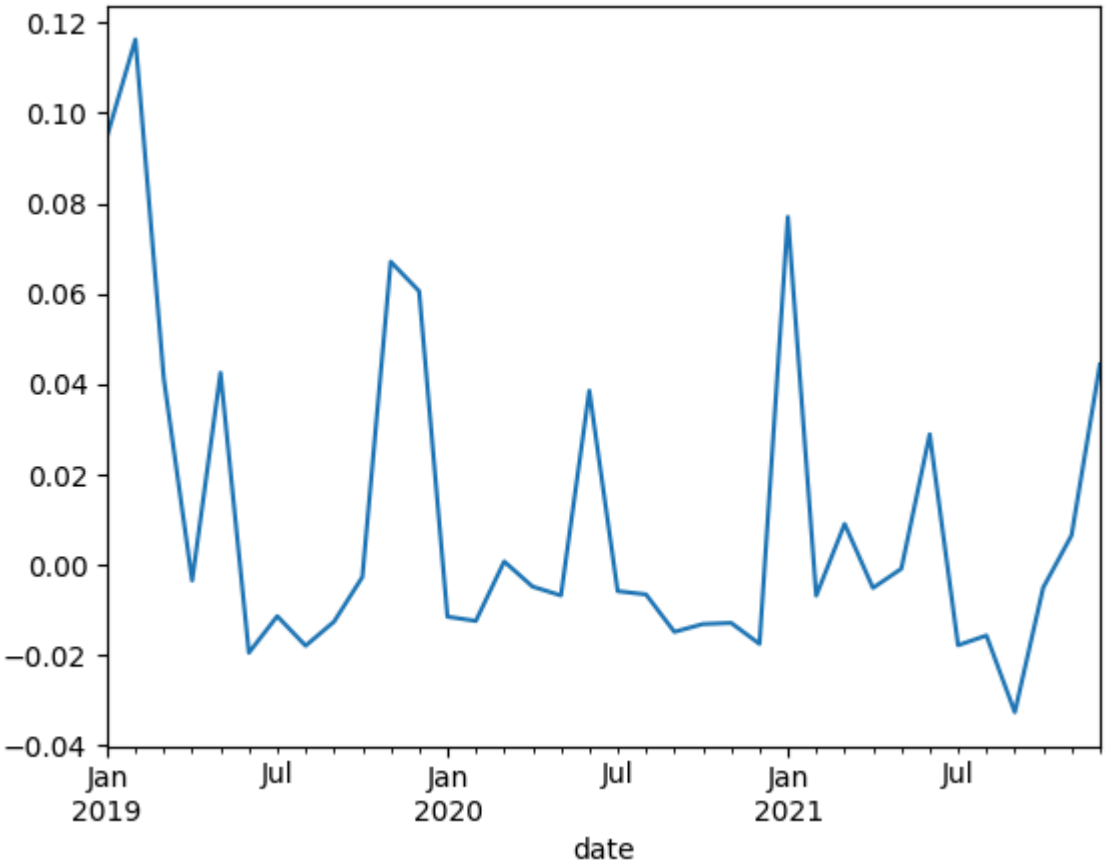
date	2019-01-01	2019-02-01	2019-03-01	2019-04-01	2019-05-01	2019-06-01	2019-07-01	201
feature_id								
0	0.094833	0.116266	0.041275	-0.003463	0.042498	-0.019456	-0.011366	-0.0
1	0.061652	0.132942	0.008162	-0.005530	0.063989	-0.009637	-0.006807	-0.0
2	0.094250	0.063221	-0.009423	-0.009157	0.107729	-0.015520	-0.013755	-0.0
3	0.109927	0.003998	-0.010714	-0.009956	0.096144	-0.013231	-0.009510	-0.0
4	0.086098	0.007734	0.005219	-0.010751	0.096271	-0.015009	-0.013761	-0.0
...	...	...	...	...	...	...	...	
101	0.151086	0.272270	0.431394	0.375502	0.379856	0.336836	0.302526	0.1
102	0.159619	0.248704	0.326311	0.318180	0.330156	0.321391	0.303998	0.1
103	0.146223	0.278045	0.417649	0.363990	0.370138	0.306579	0.275267	0.1
104	0.217346	0.402631	0.360071	0.427471	0.334912	0.348880	0.328537	0.3
105	0.217331	0.336104	0.353571	0.416882	0.349251	0.380856	0.363016	0.3

106 rows × 36 columns

Let's visualize one of the time series for an example point:

In [ ]:
```
pivoted_df.iloc[0].plot()
```

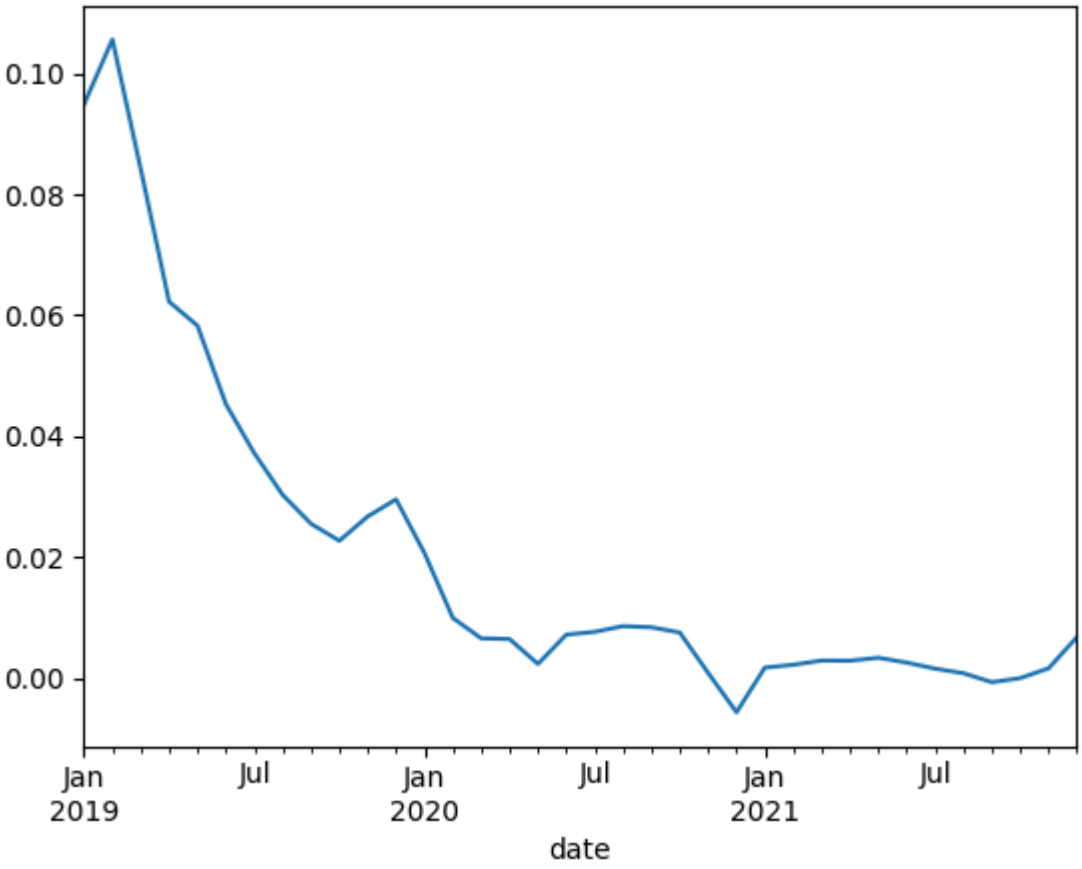
Out[ ]:
```
<Axes: xlabel='date'>
```



Let's check what the result would be if we applied the moving average to smooth the series:

In [ ]:
```
pivoted_df.iloc[0].rolling(window=12, min_periods=1).mean().plot()
```

Out[ ]:
```
<Axes: xlabel='date'>
```



We then apply to all points:

In [ ]:
```
df_rolling_mean = pd.DataFrame()
for i,row in pivoted_df.iterrows():
  df_aux = pivoted_df.iloc[i].rolling(window=12, min_periods=1).mean()
  df_rolling_mean = pd.concat([df_rolling_mean,df_aux], axis=1)
```

In [ ]:
```
df_rolling_mean = df_rolling_mean.T.reset_index(drop=True)
```

Then we convert it to the required format:

In [ ]:
```
time_series_dataset = to_time_series_dataset(df_rolling_mean)
```

Let's install sktime:

In [ ]:
```
!pip install sktime
```

```
Collecting sktime
  Downloading sktime-0.36.0-py3-none-any.whl.metadata (34 kB)
Requirement already satisfied: joblib<1.5,>=1.2.0 in /usr/local/lib/python3.11/di
st-packages (from sktime) (1.4.2)
Requirement already satisfied: numpy<2.3,>=1.21 in /usr/local/lib/python3.11/dist
-packages (from sktime) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packag
es (from sktime) (24.2)
Requirement already satisfied: pandas<2.3.0,>=1.1 in /usr/local/lib/python3.11/di
st-packages (from sktime) (2.2.2)
Collecting scikit-base<0.13.0,>=0.6.1 (from sktime)
  Downloading scikit_base-0.12.0-py3-none-any.whl.metadata (8.5 kB)
Requirement already satisfied: scikit-learn<1.7.0,>=0.24 in /usr/local/lib/python
3.11/dist-packages (from sktime) (1.6.1)
Requirement already satisfied: scipy<2.0.0,>=1.2 in /usr/local/lib/python3.11/dis
t-packages (from sktime) (1.14.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
1/dist-packages (from pandas<2.3.0,>=1.1->sktime) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-pac
kages (from pandas<2.3.0,>=1.1->sktime) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-p
ackages (from pandas<2.3.0,>=1.1->sktime) (2025.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/
dist-packages (from scikit-learn<1.7.0,>=0.24->sktime) (3.6.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-package
s (from python-dateutil>=2.8.2->pandas<2.3.0,>=1.1->sktime) (1.17.0)
Downloading sktime-0.36.0-py3-none-any.whl (36.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 36.9/36.9 MB 30.3 MB/s eta 0:00:00
Downloading scikit_base-0.12.0-py3-none-any.whl (141 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 141.5/141.5 kB 8.3 MB/s eta 0:00:00
Installing collected packages: scikit-base, sktime
Successfully installed scikit-base-0.12.0 sktime-0.36.0
```

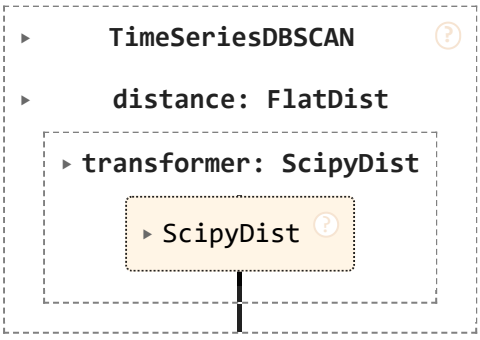
Then we can apply TimeSeriesDBSCAN:

In [ ]:
```python
from sktime.clustering.dbscan import TimeSeriesDBSCAN
from sktime.dists_kernels import FlatDist, ScipyDist

eucl_dist = FlatDist(ScipyDist())
clst = TimeSeriesDBSCAN(distance=eucl_dist, eps=0.2)
```

In [ ]:
```python
clst.fit(time_series_dataset)
```

Out[ ]:



We get the labels and join them with the original GeoDataFrame:

In [ ]:
```python
labels = clst.labels_
```

In [ ]:
```
gdf_full['classe'] = labels
```

Finally we present with Folium:

In [ ]:
```
resultmap = folium.Map(location=[gdf_full.geometry.y.values[0],gdf_full.geometry
basemaps['Google Satellite Hybrid'].add_to(resultmap)

latitudes = list(gdf_full.geometry.y.values)
longitudes = list(gdf_full.geometry.x.values)
labels = list(gdf_full['classe'].values)
for lat, lng, label in zip(latitudes, longitudes, labels):
  if label == 0:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='red')
     ).add_to(resultmap)
  elif label == 1:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='blue')
     ).add_to(resultmap)
  elif label == 2:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='black')
     ).add_to(resultmap)
  else:
    folium.Marker(
      location = [lat, lng],
      popup = str(label),
      icon = folium.Icon(color='green')
     ).add_to(resultmap)


vis_params = {'min': 0, 'max': 1}# Add the elevation model to the map object.
resultmap.add_child(folium.LayerControl())

# Display the map.
display(resultmap)
```

Thank you! See you in the next Chapter!

References:

https://sia-ai.medium.com/time-series-clustering-b84bcaaa63ac

https://towardsdatascience.com/time-series-clustering-deriving-trends-and-archetypes-from-sequential-data-bb87783312b4/

https://www.geeksforgeeks.org/time-series-clustering-techniques-and-applications/

https://tslearn.readthedocs.io/en/stable/user_guide/clustering.html

https://medium.com/@katser/a-list-of-python-packages-for-time-series-analysis-c883bcadcc58