Podstawowe operacje na danych wektorowych i rastrowych

Krótkie informacje na temat użytych bibliotek

Do realizacji zadań korzystających z danych rastrowych i wektorowych będziemy korzystać z kilku bibliotek Python. Na start chciałem krótko je omówić, przedstawić potencjalne alternatywy i wytłumaczyć się czemu akurat korzystamy z takiego zestawu technologii, a nie innego.

Dane wektorowe

Do pracy z danymi wektorowymi użyjemy znanej wam już trochę biblioteki GeoPandas. Jeżeli chodzi o przetwarzanie danych wektorowych wybór (przynajmniej jeżeli chodzi o poważne biblioteki) nie jest zbyt duży.

Największym otwartym pakietem do pracy z danymi wektorowymi jest OGR będący komponentem pakietu GDAL. OGR rozwijany przez OSGeo umożliwia podstawowe operacje na danych wektorowych. Do jego głównych zalet należy przede wszystkim ogromny arsenał obsługiwanych formatów danych. OGR świetnie się sprawdza jeżeli chodzi o wczytanie danych, podstawową przestrzenną filtrację, manipulowanie atrybutami. Jeżeli chodzi o obliczenia na geometrii - tutaj funkcjonalność raczej jest dość ograniczona. Głównym powodem z jakiego nie korzystamy bezpośrednio z OGR w Python jest jego implementacja. OGR oryginalnie zaimplementowany jest w języku C++. Dostępny jest Pythonowy interfejs, jest on jednak dość toporny i wymaga bardzo specyficznego podejścia i dużej uwagi. Z tego powodu rzadko korzysta się z "gołego" OGR.

Żeby rozwiązać część problemów z OGR zaimplementowany został pakiet Python nakładany na OGR - Fiona. Co do zasady biblioteka zawiera mnóstwo funkcji i struktur danych, które mają ułatwić pracę z OGR. Z tego powodu nie spotkamy tutaj raczej żadnej nowej funkcjonalności względem OGR.

No i tutaj dochodzimy do naszego zawodnika. Chwilowo najbardziej zaawansowaną i najczęściej używaną biblioteką do pracy z danymi wektorowymi jest GeoPandas. Jako, że już z nim trochę pracowaliście, nie będę się specjalnie nad nim rozwodził. Może w skrócie na co dodatkowo pozwala GeoPandas:

- Zapewnia strukturę danych opartą o Pandas, więc jest bardzo Python-friendly
- Korzysta z OGR do wczytywania danych - dostępne są dwa silniki wczytywania (Fiona oraz pyogrio - ich implementacja, w teorii trochę szybsza)
- Umożliwia pracę z geometrią korzystając z biblioteki Shapely (będącej nakładką na bibliotekę GEOS - również całkiem poważne rozwiązanie)
- Obsługuje standardowe układy współrzędnych z wykorzystaniem PyPROJ

Dane rastrowe

Tutaj temat ma kilka poziomów...

Generalnie do wielu z operacji będziemy wykorzystywać tylko tablice w NumPy. NumPy jest szybki i prosty - tu nie powinno być problemów. W przypadku kiedy chcemy pracować z danymi rastrowymi z wykorzystaniem trochę poważniejszych funkcji musimy sięgnąć po trochę bardziej specjalistyczne rozwiązania. Tutaj chciałbym polecić 3 (wg mnie) najlepsze rozwiązania:

- OpenCV - biblioteka najprostsza w konstrukcji, ale zapewniająca mnóstwo funkcji. Z niej będziemy korzystać na zajęciach, ponieważ będziemy wykonywać tylko proste operację czysto rastrowe, a akurat OpenCV ma bardzo prosty i łatwy do zrozumienia interfejs.
- Pillow - biblioteka zbliżona do OpenCV. Biblioteka umożliwia wczytywanie rastrów, podstawowe manipulacje takie jak przesuwanie, przycinanie obracanie itd. Wiele osób korzysta z niej ze względu na bardzo wygodny interfejs obiektowy stworzony baardzo pod Python. Jak porównać Pillow do OpenCV? Cytując: "Pillow używasz jeżeli chcesz przyciąć i obrócić, ewentualnie trochę odfiltrować obrazek. OpenCV wykorzystujesz jeżeli budujesz robota, który ma widzieć"
- Scikit-Image - Jeżeli szukamy czegoś w miarę do wszystkiego Skimage będzie najlepszym wyborem. Poza wczytywaniem rastrów (opartym o mechanizmy z Pillow i OpenCV) zawiera też bardzo dużo algorytmów przetwarzania obrazów takich jak binaryzacja, konwersje kolorów. Zdecydowanie w porównaniu do Pillow jest to bardzo rozbudowana biblioteka, która bardzo dobrze sprawi się jeżeli jesteśmy krok przed budowaniem robota. Ze Skimage nie będziemy korzystać, ponieważ według mnie jest to znaczny overkill. Będziemy potrzebować tylko kilku funkcji, a Skimage wymaga jednak chwili wprowadzenia. Tym nie mniej zachęcam żeby poznać tą bibliotekę, jeżeli ktoś w wolnym czasie lubi bawić się w obrabianie zdjęć.

Ostatnim tematem, który należy poruszyć jest obsługa georastrów. O tym czym się one różnią od zwykłych rastrów porozmawiamy dokładnie na zajęciach. Żeby tylko nadać kontekst:

1. Georastry muszą mieć dużo dodatkowych metadanych, których w zwykłym rastrze nie zapiszemy bo formaty tego nie przewidują (a OpenCV i tak nie umiałoby ich odczytać)
2. Rastry używane w GIS często zawierają więcej danych (np. więcej kanałów, piramidy obrazów, kafelki), które trzeba obsłużyć. Z tych powodów potrzebujemy narzędzia, które obsłuży przynajmniej odczyt rastrów geo. Tutaj podobnie jak w przypadku danych wektorowych stawiamy na OSGeo i pakiet GDAL. Nie znajdziemy za bardzo alternatyw, które obsługują rastry przestrzenne w tylu formatach do GDAL. Jedyną wadą GDAL podobną jak w przypadku OGR jest bardzo nieoczywiste API.

Niejako odpowiednikiem GeoPandas w przypadku GDAL jest RasterIO - biblioteka wykorzystująca GDAL do wczytywania, która dodatkowo zapewnia dużo różnych funkcji (np. testowane już przez was maskowanie), które w innym wypadku musielibyśmy implementować sami. Przy danych rastrowych skorzystamy zarówno z Rasterio i GDAL. Tam gdzie nie będzie to wymagało specjalnej gimnastyki wykorzystamy GDAL, ponieważ pozwala on dobrze zobaczyć jak georastry są zbudowane pod maską. Jeżeli będziemy potrzebować zrobić coś bardziej złożonego, skorzystamy z RasterIO żeby zaoszczędzić czas.

Tutorial

Wczytywanie danych rastrowych

Na początek przejdziemy przez proces wczytywania rastra i omówimy sobie kluczowe momenty i elementy.

Zacznijmy od odczytywania rastra. Tak jak wspominałem używamy GDAL. Od razu po imporcie sugeruję też korzystać z dodatkowego mechanizmu wyjątków GDAL - domyślnie jeżeli funkcja GDAL napotka błąd to nie przerwie to wykonania programu. Przykładowo:

Wczytujemy raster, ale wczytywanie nie powiodło się, więc zmienna z rastrem po wczytaniu ma wartość Null. O błędzie dowiadujemy się dopiero przy próbie jej użycia, albo z terminala gdzie wypisany zostanie stacktrace erroru. Żeby łatwiej kontrolować program polecam skorzystać więc z `gdal.UseExceptions()` ponieważ umożliwi ono łatwiejsze debugowanie programu.

```
from osgeo import gdal

gdal.UseExceptions()
```

Po imporcie GDAL możemy zacząć pracę z danymi rastrowymi. Wczytajmy więc georaster. I przyjrzyjmy się trochę co mamy w nim zapisane.

```
from typing import Union
from pathlib import Path

def read_spatial_raster(path: Union[str, Path]) -> gdal.Dataset:
    dataset = gdal.Open(str(path))
    assert dataset is not None, "Read spatial raster returned None"
    return dataset

raster_file = "georaster.tif"
raster_dataset = read_spatial_raster(raster_file)

print("Natywny układ współrzędnych rastra:",
raster_dataset.GetProjection())
print("Patametry transformacji z układu XY do układu UV rastra:",
raster_dataset.GetGeoTransform())
print("Liczba kanałów:", raster_dataset.RasterCount)
print("Wymiary rastra w pikselach (szerokość x wysokość):",
[raster_dataset.RasterXSize, raster_dataset.RasterYSize])
```

Sam Dataset nie zawiera jeszcze danych (komórek rastra). Dataset może zawierać wiele kanałów, musimy więc wczytać je ręcznie żeby dalej z nimi pracować. W tym celu musimy stworzyć referencję do obiektu Band i wczytać go sobie do NumPy.

```
import numpy as np

def read_raster_band(dataset: gdal.Dataset, band_number: int) ->
```

```
gdal.Band:
    assert 0 < band_number <= dataset.RasterCount, f"Band number
{band_number} is invalid for raster with {dataset.RasterCount} bands."
    band = dataset.GetRasterBand(band_number)
    assert band is not None, f"Unable to read band {band_number}"
    return band

def read_band_as_array(band: gdal.Band) -> np.ndarray:
    array = band.ReadAsArray()
    array = np.copy(array)  # To make sure we do not get memory errors
    return array

band = read_raster_band(raster_dataset, 1)
array = read_band_as_array(band)

print("Przyjęta wartość NoData dla rastra:", band.GetNoDataValue())
print("Typ liczbowy komórek rastra (wartość z Enum):", band.DataType)
print("Zakres wartości:", band.ComputeRasterMinMax())   # There are
also GetMinimum, GetMaximum

print("Wymiary tablicy", array.shape)
print("Typ liczbowy:", array.dtype)
print("Statystyki:", [array.min(), array.max()])
print("Liczba komórek NoData:", np.sum(array ==
band.GetNoDataValue()))
```

Typ liczbowy w typ przypadku to kod z enumeracji

Wizualizacja rastra

Żeby mieć pewność, że faktycznie odczytaliśmy dane poprawnie wyświetlmy je sobie tak jak w ArcGIS. Tutaj jako, że działamy na tablicach NumPy wszystkie standardowe mechanizmy powinny zadziałać. Trzeba tylko zwracać uwagę na typy wartości i ich zakresy. Większość bibliotek do wizualizacji danych w Python stworzone zostały jednak z myślą o rastrach 1 lub 3 kanałowych z wartościami 8-bitowymi. Rastry geo mogą mieć jak wiemy dowolną liczbę kanałów, a piksele mogą mieć zarówno wartości 8, 16, czy 32-bitowe całkowitoliczbowe, ale również przechowywać kanały zmiennoprzecinkowe (Float32 i Float64). Niektóre biblioteki lub formaty plików mogą tego nie akceptować, trzeba więc w razie potrzeby dostosować zawartość NumPy.

Ja do wizualizacji najbardziej lubię korzystać z imshow w matplotlib, ponieważ wspiera bez żadnych dodatkowych zabaw zoomowanie i skale barwne. Podobne mechanizmy są też dostępne w OpenCV, jeżeli ktoś chciałby eliminować zależności w kodzie.

```
%matplotlib qt
import matplotlib.pyplot as plt

def show_grayscale_matplotlib(array: np.ndarray):
    plt.imshow(array, cmap='gray')
```

```
show_grayscale_matplotlib(array)
plt.show()
```

Integracja danych wektorowych i rastrowych

Skoro wczytaliśmy raster i wszystko jest z nim w porządku, spróbujmy doczytać dane wektorowe i wyświetlić je na jego tle.

Jak pewnie się domyślacie mamy tu do czynienia z 1 lub 2 przeliczeniami współrzędnych:

1. (Opcjonalne) Przeliczenie danych wektorowych z ich układu współrzędnych do natywnego układu rastra (np. EPSG 3857 -> EPSG 2180).
2. Przeliczenie danych wektorowych z natywnego układu rastra do układu pikselowego.

Dlaczego w punkcie 2. akurat przeliczamy do układu pikselowego? Ten sposób jest po prostu łatwiejszy. Można w teorii przeliczyć piksele do układu odwzorowawczego, ale na dłuższą metę utrudni to nam pracę:

- Piksele przestaną być już kwadratowe;
- Nie mamy gwarancji, że będą one w siatce (nawet równoległoboku);
- Tracimy w pewien sposób informację o sąsiedztwie pikseli. Ogólnie to podejście w pewien sposób zamieni nasz raster w warstwę punktową o dużej liczbie obiektów.

Przejdźmy więc do przeliczenia. Przeliczenie odbywa się w oparciu o tablicę z GetGeoTransform i formuły ze strony GDAL. Podejdziemy więc do tego zagadnienia etapami.

Implementacja prosta

Zgodnie z dokumentacją i kodem źródłowym GDAL zamiana pojedynczego punktu z układu natywnego do układu pikselowego wykonywana jest w następujący sposób:

```
from typing import Tuple, List

def point_to_pixel(x: float, y: float, geotransform: List[float]) ->
Tuple[float, float]:
    c, a, b, f, d, e = geotransform
    column = (x - c) / a
    row = (y - f) / e
    return row, column  # ij convention to stay with NumPy

example_point = [501902.401, 531640.282]    # Center of a white blob
in EPSG:2180 (raster native CRS)
i, j = point_to_pixel(example_point[0], example_point[1],
raster_dataset.GetGeoTransform())
i, j

show_grayscale_matplotlib(array)
plt.scatter(j, i, s=100, c='red')    # Show point with big red dot
plt.show()
```

Implementacja NumPy

Oczywiście gdybyśmy chcieli teraz przeliczyć powyższą funkcją warstwę wektorową do układu rastra to nic nie stoi na przeszkodzie. Dla każdego obiektu wywołujemy funkcję dla każdego narożnika / punktu i misja wykonana. Jedyną wadą tego rozwiązania (z punktu widzenia Python) jest to, że kod taki będzie działał wolno. Dlaczego? Dużo by opowiadać, ale może dam mały zarys w tym linku. Dlatego proponuję jedno małe usprawnienie, żeby nasz kod bardziej korzystał z NumPy. Zamiast pojedynczych punktów będziemy przyjmować tablicę punktów (np. wszystkie wierzchołki poligonu) i wykonamy na nich obliczenia w jednym przebiegu:

```
def points_to_pixels(points: np.ndarray, geotransform: List[float]) ->
np.ndarray:
    c, a, _, f, _, e = geotransform
    columns = (points[:, 0] - c) / a
    rows = (points[:, 1] - f) / e
    pixels = np.vstack([rows, columns])
    pixels = pixels.T
    return pixels

example_points = [
    [501902.401, 531640.282],   # Still blob
    [501378.971, 531797.020],   # Some tree
    [501481.17, 532950.23]      # Field in the forest
]
example_points = np.float64(example_points)
pixels = points_to_pixels(example_points,
raster_dataset.GetGeoTransform())
show_grayscale_matplotlib(array)
plt.scatter(pixels[:, 1], pixels[:, 0], s=100, c='red')    # Show
point with big red dot
plt.show()
```

Przykład z warstwą wektorową

Przenieśmy teraz implementację do GeoPandas. Schemat wygląda mniej więcej tak:

1. Wczytujemy warstwę wektorową
2. Przeliczamy warstwę do natywnego układu rastra
3. Przeliczamy obiekty wektorowe do układu pikselowego
4. Wizualizujemy wszystko w Matplotlib

Wszystko powinno być raczej oczywiste. Żeby przetestować funkcję do wizualizacji z GeoPandas konieczne było zmienienie kolejności ij na ji (funkcja `transform_function`).

```
import geopandas as gpd
import pandas as pd
import shapely


def read_features_to_geopandas(path: Union[str, Path]) ->
```

```
gpd.GeoDataFrame:
    features = gpd.read_file(path)
    return features


def reproject_geodataframe(features: gpd.GeoDataFrame, crs: str) ->
gpd.GeoDataFrame:
    return features.to_crs(crs)


def convert_to_pixel_system(features: gpd.GeoDataFrame, geotransform:
List[float]) -> gpd.GeoDataFrame:
    def transform_function(xy: np.ndarray):
        ij = points_to_pixels(xy, geotransform)
        ji = ij[:, [1, 0]]
        return ji


    indices = features.index
    for i in indices:
        geometry = features.loc[i, "geometry"]
        geometry = shapely.transform(geometry, transform_function)  #
To make our solution work for every type of geometry
        features.loc[i, "geometry"] = geometry
    return features


features_file = "features.fgb"
features = read_features_to_geopandas(features_file)
features = reproject_geodataframe(features,
raster_dataset.GetProjection())
features = convert_to_pixel_system(features,
raster_dataset.GetGeoTransform())

fig, ax = plt.subplots()
ax.imshow(array, cmap='gray')
features.plot(ax=ax)
fig.show()
```

W tym momencie mozemy używać poligonów do badania danych rastrowych!

Analiza wartości pikseli (spektralnych)

Skoro udało nam się zintegrować dwa typy danych, spróbujmy je wykorzystać do jakiejś prostej analizy. Docelowo chcemy obliczyć statystyki wartości pikseli wewnątrz określonych poligonów. Podzielimy to zadanie na dwie części, ponieważ pierwsza pozwoli generować ładne obrazki do sprawozdania.

Wycinanie mniejszych fragmentów rastra

Jeżeli mamy wczytane dane możemy bardzo wygodnie korzystając z NumPy wycinać sobie z naszego pełnego rastra mniejsze (prostokątne) fragmenty. Załóżmy że chce wyciąć na przykład kafelek 1000x1000 pikseli w lewym górnym rogu rastra. Mogę to bez problemu zrealizować w NumPy jedną komendą:

```
fragment = array[
    0: 1000,    # 1000 rows
    0: 1000     # 1000 columns
]
show_grayscale_matplotlib(fragment)
```

Jestem więc o krok od tego, żeby w analogiczny sposób wyciąć np. piksele w obrębie jednego z moich poligonów. Muszę tylko wiedzieć w jakich zakresach kolumn i wierszy się znajduje. Jako, że moje poligony już są w układzie pikselowym muszę tylko wyciągnąć ich BBOX (polecenie `bounds` dla obiektu Shapely) i dopilnować, żeby przejść na liczby całkowite (inaczej indeks NumPy nie zadziała):

```
example_feature = features.iloc[1]  # Select sample feature from our
layer
example_polygon = example_feature["geometry"]

bounds = example_polygon.bounds
bounds = np.float64(bounds)
print("BBOX poligonu:", bounds)

bounds[:2] = np.floor(bounds[:2])
bounds[2:] = np.ceil(bounds[2:])
bounds = np.int64(bounds)
print("BBOX poligonu (integer):", bounds)

fragment = array[
    bounds[1]: bounds[3],
    bounds[0]: bounds[2]
]
show_grayscale_matplotlib(fragment)
```

Maskowanie

W powyższy sposób mogę wygodnie wycinać prostokątne fragmenty. Jeżeli chcę dodatkowo pozbyć się pikseli położonych w BBOX, ale poza moim poligonem (np. do analiz) muszę je wymasować.

Jak już wiadomo - maskowanie generalnie sprowadza się do zamiany wybranych pikseli na przyjętą przez mnie wartość, którą potem będę systematycznie pomijał w analizach. Na razie zróbmy to do celów wizualizacji - piksele poza moim poligonem zamienię na czarne.

Generalnie proces ten można jak zawsze przeprowadzić samemu - trzeba wyciąć mniejszy prostokąt, przejść przez piksele i sprawdzać czy zawierają się one w moim prostokącie.

Skorzystajmy jednak może z czegoś gotowego. Wykorzystamy moduł mask z pakietu rasterio. Omówimy przy okazji parę ważnych z punktu widzenia późniejszej pracy aspektów, więc przejdźmy przez ten proces krok po kroku.

Wyciąłem już mniejszy fragment rastra, jednak mój poligon nadal jest w układzie "dużego" rastra. Muszę więc przesunąć go w oparciu o BBOX, którym przeciąłem. Skorzystamy z gotowej funkcji w Shapely:

```
polygon_in_fragment_frame =
shapely.affinity.translate(example_polygon, -bounds[0], -bounds[1])
```

Teraz muszę zamienić mój poligon na maskę wielkości fragmentu rastra. Czym jest dokładnie maska?

NumPy zapewnia wiele ciekawych trybów indeksowania tablic. Jeden z nich pozwala wybrać elementy z tablicy wykorzystując takich samych rozmiarów tablicę typu bool - Boolean array indexing.

Na początek zróbmy prosty eksperyment - stworzymy maskę, która z pełnego rastra usunie tylko narożniki. Musimy więc:

1. Stworzyć tablicę tej samej wielkości co raster
2. Elementy które mamy wymaskować ustawić na True, a te które mają zostać na False
3. Podmienić wartości na wybrane przez nas NoData

Może najpierw przykład na prostej tablicy:

```
test_array = np.random.random(10)                    # Array of 10
random floats
mask = np.zeros(test_array.shape, dtype=np.bool_)   # Array of 10
Falses
mask[[0, 4, 6, 9]] = True   # I want to change 0, 4, 6, and 9-th
element of array

print("Moja tablica:", test_array)
print("Indeks logiczny:", test_array[mask])
print("Indeks logiczny (negacja): ", test_array[~mask]) # Cannot use
`not` for numpy arrays - use ~

test_array[mask] = -1.0
print("Moja tablica po maskowaniu:", test_array)
```

Tę samą logikę teraz zastosujmy dla rastra

```
mask = np.zeros(array.shape, dtype=np.bool_)
size = 2000
mask[0:size, 0:size] = True
mask[-size:, 0:size] = True
mask[-size:, -size:] = True
mask[0:size, -size:] = True
```

```
masked_array = np.copy(array)
masked_array[mask] = 0
show_grayscale_matplotlib(masked_array)
```

Dobrze, teraz musimy stworzyć maskę w oparciu o poligon:

```
from rasterio.features import rasterize

no_data_mask = rasterize([polygon_in_fragment_frame], fragment.shape)
no_data_mask = np.bool_(no_data_mask)
no_data_mask = ~no_data_mask    # Rasterio puts True inside polygon
masked_fragment = np.copy(fragment)
masked_fragment[no_data_mask] = 0
show_grayscale_matplotlib(masked_fragment)
```

Wyznaczanie statystyk dla obiektów (Zonal Statistics)

Jeżeli umiem wybrać piksele położone wewnątrz mojego poligonu to mogę teraz bez problemu wyznaczyć dowolne istotne dla mnie statystyki. Korzystając z poprzednio wyciętego fragmentu mogę na przykład wyznaczyć minimalną, maksymalną i średnią wartość pikseli:

```
pixel_values = fragment[~no_data_mask]  # Select just valid pixels
pixel_values, pixel_values.min(), pixel_values.max(),
pixel_values.mean()
```

Jeżeli chcę pominąć etap wycinania mogę też stworzyć maskę w jednym kroku:

```
object_mask = rasterize([example_polygon], array.shape)
object_mask = np.bool_(object_mask)
pixel_values = array[object_mask]
pixel_values, pixel_values.min(), pixel_values.max(),
pixel_values.mean()
```

Filtracja

Badanie kształtu i struktury obiektów widocznych na zdjęciach jest już zagadnieniem dalece bardziej skomplikowanym. Zagadnienia te są poruszane dokładniej na innych przedmiotach, więc tutaj ograniczę się tylko do krótkiego zarysu.

Wykrywanie kształtów, linii prostych itd. wymaga nie tylko analizy pojedynczych pikseli, ale również ich sąsiedztwa. Często nazywamy takie analizy kontekstualnymi (ponieważ potrzebują szerszy kontekst). Analizy kontekstualne pozwalają nie tylko znajdować krawędzie. Z ich pomocą możemy na przykład rozmazywać zdjęcia, uwydatniać szczegóły, rozszerzać i zwężać wybrane konfiguracje pikseli itd. Wszystkie z tych analiz bazują na jednej lub wielu złożonych operacjach filtracji rastra. Filtracja w skrócie polega na przejściu przez piksele obrazu mniejszym oknem (np. 5x5 pikseli) i wykonaniu w obrębie okna jakiejś operacji arytmetycznej (np. wybieram z 25 pikseli najjaśniejszy, odejmuje wartości pikseli po prawej od wartości pikseli po lewej). Wynik obliczeń na oknie daje nam nową wartość jasności piksela obrazu pochodnego. Temat jest szeroki, ale

zostawię w materiałach parę linków, jeżeli kogoś zainteresuje to zagadnienie. Przejdźmy natomiast do konkretnego przykładu:

Jeżeli chcę zbadać czy na moim obrazie jest jakaś krawędź (na przykład pionowa) muszę zbadać czy wartość pikseli zmienia się w konkretny sposób (piksele z lewej są jaśniejsze/ciemniejsze niż piksele z prawej). W widzeniu maszynowym takie filtry, które "znajdują" krawędzie na obrazie nazywamy filtrami krawędziowymi (edge operators, edge kernels). Jest ich parę - my skorzystamy z operatora Sobel'a. To przy okazji przykład, gdzie z pomocą przyjdą algorytmy z OpenCV.

Na początek wytnę z pełnego rastra mniejszy fragment

```
fragment = array[8000:9000, 6000:7000]
```

Wykrywanie krawędzi jest bardzo wyczulone na szumy widoczne na obrazie, więc bardzo często (jeżeli chcemy wykrywać krawędzie większych obiektów) przed wykrywaniem jeszcze rozmazujemy obraz - to pozwala zredukować szum (ziarnistość)

```
import cv2

blurred = cv2.GaussianBlur(fragment, (7,7), sigmaX=0, sigmaY=0)
show_grayscale_matplotlib(np.hstack([fragment, blurred]))
```

Teraz już możemy przejść do etapu faktycznego uwydatniania krawędzi

```
sobelxy = cv2.Sobel(src=blurred, ddepth=cv2.CV_64F, dx=1, dy=1,
ksize=7)
plt.imshow(sobelxy, cmap='gray')
```

Jak widać na pochodnym obrazie teraz jasne piksele pokazują gdzie w oryginalnym obrazie piksele zmieniały barwy tak, że tworzyły krawędzie

Binaryzacja i zamiana na poligony

Ostatnim elementem przetwarzania rastrów, który chciałbym poruszyć w tym poradniku to jeszcze binaryzacja i zamiana pikseli na poligony wektorowe.

Zadaniem teledetekcji jest znajdowanie obiektów na obrazach. W wielu przypadkach chcemy, żeby ostatecznym produktem naszego przetwarzania były więc wykrycia obiektów w postaci wektorowej (szukam budynków, chcę poligony budynków). Spróbujmy więc na paru prostych przykładach wykonać taką operację w Python.

Zacznijmy od rozgrzewkowego przykładu. Stworzę sobie raster na którym będą 4 prostokąty w różnych barwach, które następnie zamienię na postać wektorową z wykorzystaniem RasterIO. Zaczynam więc od rastra i prostokątów:

```
size = (600, 800)
test_image = np.zeros(size, dtype=np.uint8)
test_image[10:70, 25:90] = 80
```

```
test_image[300:390, 200:250] = 120
test_image[500:520, 500:700] = 200
test_image[300:320, 500:700] = 200
show_grayscale_matplotlib(test_image)
```

Teraz muszę z rastra stworzyć obiekty wektorowe. Oznaczałoby to, że muszę znaleźć na obrazie grupy pikseli w tych samych barwach, następnie wyznaczyć punkty będące ich obrysem i stworzyć z tego poligon. Na szczęście w RasterIO jest do tego gotowa funkcja, więc nie będziemy tego implementować sami.

```
from rasterio.features import shapes

def segment_image(image: np.ndarray):
    shapes_from_image = shapes(image)
    shapes_from_image = [{'properties': {'raster_val': v}, 'geometry':
s} for s,v in shapes_from_image]
    shapes_from_image =
gpd.GeoDataFrame.from_features(shapes_from_image)
    return shapes_from_image

rectangles = segment_image(test_image)
rectangles.plot("raster_val")
```

Mamy sukces, ale z małym błędem. Jak widać cały obszar mojego rastra jest wypełniony poligonami. To dlatego, że domyślnie każda unikatowa barwa piksela zostanie zamieniona w poligon. Mógłbym teraz obejść ten problem na dwa sposoby - usunąć poligon odpowiadający mojemu NoData (czyli 0):

```
rectangles[rectangles["raster_val"] != 0].plot("raster_val")
```

W wielu przypadkach jednak bardziej może nam się przydać dodatkowy parametr z funkcji shapes, pozwalający na przekazanie maski, które piksele w ogóle bierzemy pod uwagę do analiz. To może znacznie przyśpieszyć poligonizację w przypadku większych rastrów.

```
def segment_image_with_mask(image: np.ndarray, include_mask:
np.ndarray):
    shapes_from_image = shapes(image, include_mask)
    shapes_from_image = [{'properties': {'raster_val': v}, 'geometry':
s} for s,v in shapes_from_image]
    shapes_from_image =
gpd.GeoDataFrame.from_features(shapes_from_image)
    return shapes_from_image

rectangles = segment_image_with_mask(test_image, test_image != 0)
rectangles.plot("raster_val")
```

Doskonale. Teraz zgodnie z oczekiwaniami dostałem tabelę z 4 obiektami wektorowymi. Możemy przejść do przykładu na właściwych danych. Załóżmy następujący scenariusz: moje

badania wykazały, że obiekty interesującej mnie klasy zawsze mają barwę pikseli z zakresu od 10 do 20 (szukam cieni na zdjęciu). Korzystając więc ze zdobytej już wiedzy mogę teraz zaimplementować prosty algorytm klasyfikacji:

```
fragment = array[8000:9000, 6000:7000]
my_class_mask = (fragment >= 10) & (fragment <= 20)  # For and on
Numpy arrays we cannot use `and` - & instead
my_class_mask = np.uint8(my_class_mask)
show_grayscale_matplotlib(my_class_mask)

detected_objects = segment_image_with_mask(my_class_mask,
my_class_mask)
detected_objects["id"] = np.int64(detected_objects.index)
detected_objects.plot("id")
```

Jeżeli uznałbym, że teraz chce dokonać dodatkowej filtracji obiektów (np. na podstawie ich pola powierzchni) to mogę skorzystać z Shapely i iść z moim algorytmem dalej. Ale to już zostawię do implementacji w ramach samokształcenia.

Na zakończenie pokażę tylko jeszcze, że ten sam proces mogę też przeprowadzić na rastrach będących wynikiem wykrywania krawędzi:

```
sobelxy_normalized = np.abs(sobelxy / sobelxy.max())
edges = sobelxy_normalized > 0.3    # I set the threshold manually to
0.3
edges = np.uint8(edges)
show_grayscale_matplotlib(edges)

edges_polygons = segment_image_with_mask(edges, edges)
edges_polygons.plot()
```

Do poczytania i pooglądania

- Wprowadzenie do GeoPandas
- Oficjalny tutorial GeoPandas
- Galeria przykładów wykorzytania GDAL i OGR w Python
- Poradnik na co uważać korzystając z GDAL/OGR w Python
- Dokumentacja biblioteki Fiona
- Dokumentacja biblioteki RasterIO
- Przykłady podobnych operacji jak nasze w RasterIO
- Filmik pokazujący działanie filtracji obrazów
- Filmik pokazujący jak działa filtracja operatorem Sobel'a