Advanced Spatial Analyses: Final Assignment

Flood delineation using machine learning

By Koen van Wiggen

Import packages

```
import numpy as np
import geopandas as gpd
import matplotlib.pyplot as plt

import rasterio
from rasterio.merge import merge
from rasterio.windows import Window
from rasterio.enums import Resampling
from rasterio.features import rasterize


import sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import f1_score, precision_score, recall_score
from sklearn.model_selection import train_test_split

import os
from google.colab import drive

import logging
logging.getLogger("rasterio").setLevel(logging.ERROR)
```

Making sure I'm working with the right working directory.

```
drive.mount('/content/drive')
os.chdir('/content/drive/My Drive/Advanced Spatial Analyses/Final_Assignment_ASA')
print("Current working directory: ", os.getcwd())
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Current working directory:  /content/drive/My Drive/Advanced Spatial Analyses/Final_Assignment_ASA
```

Loading the data

To answer question 1-3 I've to download Sentinel-2 satellite imagery that is suitable for this assignment. I've found data concerning a flood in Greece in October 2023. However, these data consists of multiple datasets that I need to transform into one datasets before continuing to the next exercises. The goal of this first part of my code is to create an image such as Figure 1 down below.

Figure 1

Loading the figures

While downloading the data it came to attention that there are some clouds on the pictures. Later on, I'm gonna use rasterio.open to open the images and create the plot. By doing so, every pixel is assigned a value between approximately 10 and 255. Because the pixels in the clouds are assigned a very high value, the other pixels are assigned a relatively low value. Therefore, I've created the code down below that gives the clouds a value that is the same as the mean of that dataframe (red, green or blue color). Furthermore, the data is normalized based on its the minimum and maximum value. This ensures that if we have made a cut in the picture the values assigned to the pixels are based on the data in this cut, and not on the whole data.

```
def normalize_cut(data, cloud_threshold):
    # If the data has a value larger than cloud_threshold it will be transformed
```

```
    # into the mean of the data.
    normalized_data = np.where(data > cloud_threshold, np.mean(data), data)
    cut_min = np.min(normalized_data)
    cut_max = np.max(normalized_data)
    normalized_data = (normalized_data - cut_min) / (cut_max - cut_min) * 255
    # Ensure that the values are between [0, 255].
    normalized_data = np.clip(normalized_data, 0, 255)
    # Ensure that the resulting values are integers.
    normalized_data = normalized_data.astype(np.uint8)
    return normalized_data
```

The function down below is made to create plots of my raster data.

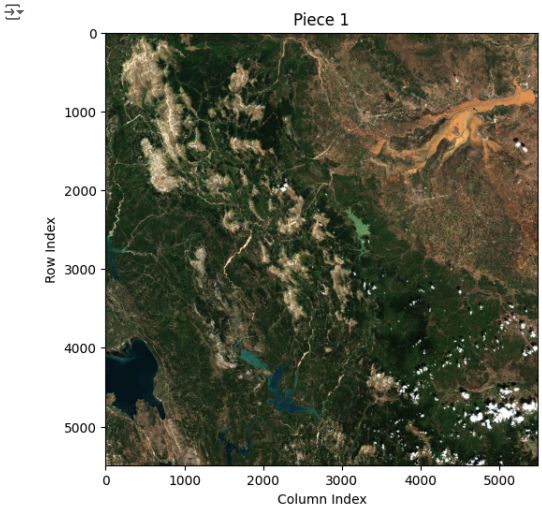
```
# Code to plot the rasters where we can adjust the title.
def plot_raster(data, title):
    plt.figure(figsize=(10, 6))
    plt.title(title)
    plt.imshow(data)
    plt.xlabel("Column Index")
    plt.ylabel("Row Index")
    plt.show()
```

Now we have a functions that adjusts the pixel values if necessary and plots the data, we can load the figures. The function down below is based on the function that was used in the assignment 4 (Optimizing the Dutch energy grid using weather data). It loads the data, cuts the figure in a certain window if necessary, renormalizes the data based on this cut and concerning the clouds, resamples the data and stores a figure of this resampled data.

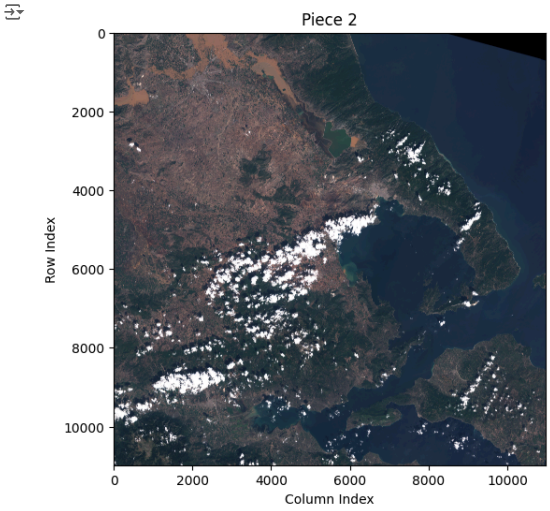
```
def resample(text1, method, text2, factor, adjust_window, start_col, start_row, width, height, adjust_cut, cloud_threshold):
    with rasterio.open(text1) as raster_src:
        # Define the window based on input.
        if adjust_window:
            window = Window(
                col_off=start_col,
                row_off=start_row,
                width=width,
                height=height
            )
        else:
            window = Window(
                col_off=0,
                row_off=0,
                width=raster_src.width,
                height=raster_src.height
            )

        # Read the data for the specified window.
        data_red = raster_src.read(1, window=window)
        data_green = raster_src.read(2, window=window)
        data_blue = raster_src.read(3, window=window)

        # Apply normalization if adjust_cut is True.
        if adjust_cut:
            data_red = normalize_cut(data_red, cloud_threshold)
            data_green = normalize_cut(data_green, cloud_threshold)
            data_blue = normalize_cut(data_blue, cloud_threshold)

        # Calculate the new dimensions for resampling.
        new_width = window.width // factor
        new_height = window.height // factor

        # Resample the data.
        resampled_data_red = raster_src.read(1, out_shape=(new_height, new_width),
            resampling=method, window=window)
        resampled_data_green = raster_src.read(2, out_shape=(new_height, new_width),
            resampling=method, window=window)
        resampled_data_blue = raster_src.read(3, out_shape=(new_height, new_width),
            resampling=method, window=window)

        # Apply normalization to resampled data if adjust_cut is True.
        if adjust_cut:
            resampled_data_red = normalize_cut(resampled_data_red, cloud_threshold)
            resampled_data_green = normalize_cut(resampled_data_green, cloud_threshold)
            resampled_data_blue = normalize_cut(resampled_data_blue, cloud_threshold)

        # Connect the separate red, green and blue values back together.
        original_data = np.dstack((data_red, data_green, data_blue))
        resampled_data = np.dstack((resampled_data_red, resampled_data_green, resampled_data_blue))

        # Adjust the transform for the window.
        window_transform = raster_src.window_transform(window)
        scaled_transform = window_transform * window_transform.scale(
            (window.width / new_width),
            (window.height / new_height))

        # Update the profile for the output file.
        resampled_crs = raster_src.profile.copy()
        resampled_crs.update({"width": new_width, "height": new_height,
            "transform": scaled_transform})

    # Move axis for saving the resampled data as output of this function.
    resampled_data2 = np.moveaxis(resampled_data.copy(), -1, 0)

    # Create the output file with the updated profile.
    with rasterio.open(text2, "w", **resampled_crs) as dst:
        dst.write(resampled_data2)

    return original_data, resampled_data
```

Let's see what we are working with. I'm now just plotting some figures without rescaling or cutting the image, and I'm not making adjustments for the clouds.

```
Greece0_before, _ = resample('Greece_image.jp2', Resampling.average, 'Greece_adjusted.jp2', 1, False, 0, 0, 0, 0, False, 0)
plot_raster(Greece0_before, 'Piece 1')
```



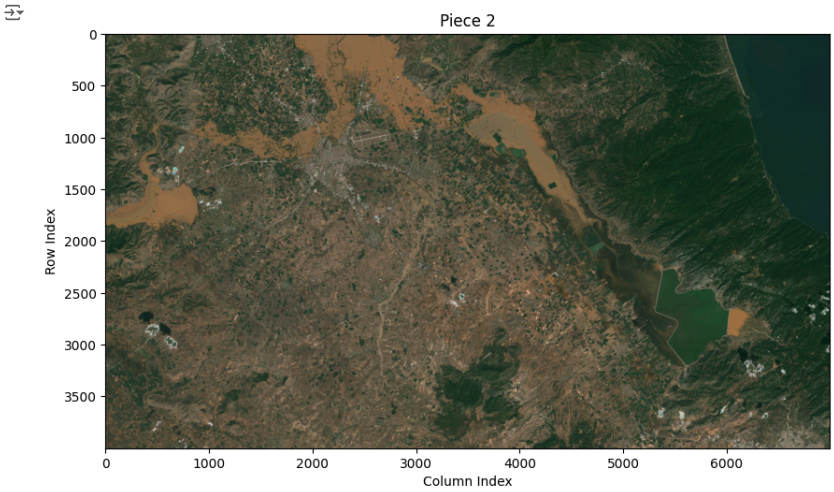

```
Greece1_before, _ = resample('Greece_image1.jp2', Resampling.average, 'Greece_adjusted1.jp2', 1, False, 0, 0, 0, 0, False, 0)
plot_raster(Greece1_before, 'Piece 2')
```



These two figures only show the bottom left and right of Figure 1. However we can clearly see that we should make adjustments for the clouds and that we only need a part of these figures to recreate Figure 1. First, I'm only gonna make appropriate adjustments for the clouds and window sizes of the figures. Then, I'm gonna merge these pieces together (six in total) to make one large figure without clouds. To see all these subplots, remove # down below. For now, I'm only showing how this second figure changes. Note: I'm aware that I also could have created one large figure without clouds first and then change the window of the figure at once. However, because I'm working with such large datasets this approach slows down the processing time of the code a lot.

```
# Piece 1
Greece0_before, _ = resample('Greece_image.jp2', Resampling.average, 'Greece_adjusted.jp2', 1, False, 0, 0, 0, 0, False, 0)
#plot_raster(Greece0_before, 'Piece 1')

# Adjusted
width0 = Greece0_before.shape[1]
Greece0_before, _ = resample('Greece_image.jp2', Resampling.average, 'Greece_adjusted.jp2', 1, True, 3000, 0, width0 - 3000, 2000, True, 254)
#plot_raster(Greece0_before, 'Piece 1')

# Piece 2
#Greece1_before, _ = resample('Greece_image1.jp2', Resampling.average, 'Greece_adjusted1.jp2', 1, False, 0, 0, 0, 0, False, 254)
#plot_raster(Greece1_before, 'Piece 2')

# Adjusted
Greece1_before, _ = resample('Greece_image1.jp2', Resampling.average, 'Greece_adjusted1.jp2', 1, True, 0, 0, 7000, 4000, True, 254)
plot_raster(Greece1_before, 'Piece 2')

# Piece 3
#Greece2_before, _ = resample('Greece_image2.jp2', Resampling.average, 'Greece_adjusted2.jp2', 1, False, 0, 0, 0, 0, False, 254)
#plot_raster(Greece2_before, 'Piece 3')

# Adjusted
Greece2_before, _ = resample('Greece_image2.jp2', Resampling.average, 'Greece_adjusted2.jp2', 1, True, 0, 3500, 3500, 1500, True, 254)
#plot_raster(Greece2_before, 'Piece 3')

# Piece 4
Greece3_before, _ = resample('Greece_image3.jp2', Resampling.average, 'Greece_adjusted3.jp2', 1, False, 0, 0, 0, 0, False, 254)
#plot_raster(Greece3_before, 'Piece 4')

# Adjusted
height_3 = Greece3_before.shape[0]
Greece3_before, _ = resample('Greece_image3.jp2', Resampling.average, 'Greece_adjusted3.jp2', 1, True, 0, 4800, 3500, height_3 - 4800, True, 254)
#plot_raster(Greece3_before, 'Piece 4')

# Piece 5
Greece4_before, _ = resample('Greece_image4.jp2', Resampling.average, 'Greece_adjusted4.jp2', 1, False, 0, 0, 0, 0, False, 254)
#plot_raster(Greece4_before, 'Piece 5')

# Adjusted
height_4 = Greece4_before.shape[0]
Greece4_before, _ = resample('Greece_image4.jp2', Resampling.average, 'Greece_adjusted4.jp2', 1, True, 3000, 4000, width0 - 3000, height_4 - 4000, True, 254)
#plot_raster(Greece4_before, 'Piece 5')

# Piece 6
```

```
#Greece5_before, _ = resample('Greece_image5.jp2', Resampling.average, 'Greece_adjusted5.jp2', 1, False, 0, 0, 0, 0, False, 254)
#plot_raster(Greece5_before, 'Piece 6')

# Adjusted
Greece5_before, _ = resample('Greece_image5.jp2', Resampling.average, 'Greece_adjusted5.jp2', 1, True, 3000, 3500, width0 - 3000, 500, True, 254)
#plot_raster(Greece5_before, 'Piece 6')
```

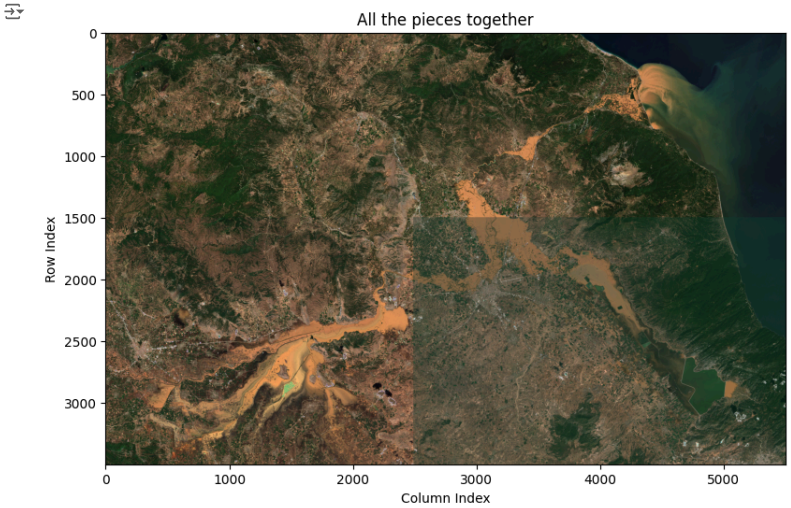


As shown in the figure above, the function to assign different values to pixels with clouds works properly. I'm gonna use the created figures to create the large figure. First, the different figures will be merged in one large figure (areas appearing in multiple pieces are considered only once). Then, this large figure will be saved.

```
def mosaic_rasters(output_path, *input_files):
    # Read the images subsequently.
    src_files = [rasterio.open(file) for file in input_files]

    # Merge the rasters, automatically ensures no overlap.
    mosaic_data, mosaic_transform = merge(src_files)

    # Update the data for the output file.
    out_meta = src_files[0].meta.copy()
    out_meta.update({
        "driver": "JP2OpenJPEG",
        "height": mosaic_data.shape[1],
        "width": mosaic_data.shape[2],
        "transform": mosaic_transform,
        "count": mosaic_data.shape[0],
    })

    # Create the output file.
    with rasterio.open(output_path, "w", **out_meta) as dest:
        dest.write(mosaic_data)

    mosaic_data = np.moveaxis(mosaic_data[:3], 0, -1)

    # Close all the open raster files.
    for src in src_files:
        src.close()

    return mosaic_data
```

Making a plot of all the pieces together.

```
final_before = mosaic_rasters("Greece_mosaic_before.jp2", "Greece_adjusted.jp2",
                              "Greece_adjusted1.jp2", "Greece_adjusted2.jp2",
                              "Greece_adjusted3.jp2", "Greece_adjusted4.jp2", "Greece_adjusted5.jp2",)
# Plot the result.
plot_raster(final_before, 'All the pieces together')
```



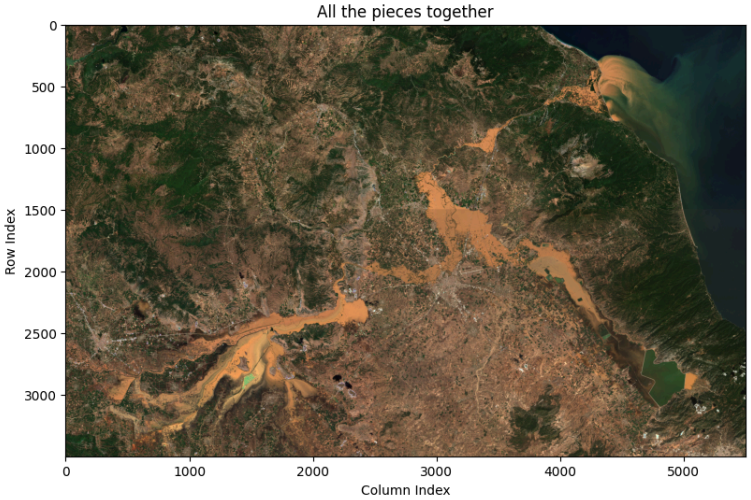
By looking at this large figure it stands out that, although the values have been renormalized based on the cut, the bottom right piece still has a slightly different color in comparison with the other pieces. The reason behind this is the large amount of clouds in this piece by which the

values of all the pixels are slightly different. To solve this, the following function can be used. It uses three files as input, the picture that should be adjusted, a picture that shows how it should be adjusted (darker, lighter, etc.) and the output path to solve this new image.

```
def adjust_colors(target_path, reference_path, output_path):
    with rasterio.open(target_path) as target, rasterio.open(reference_path) as ref:
        # Read the colors of the target and reference images.
        target_data = target.read()
        ref_data = ref.read()

        # Compute the statistics of both images.
        target_means = target_data.mean(axis=(1, 2))
        target_stds = target_data.std(axis=(1, 2))

        ref_means = ref_data.mean(axis=(1, 2))
        ref_stds = ref_data.std(axis=(1, 2))

        # Adjust the target statistics towards the reference statistics.
        adjusted_data = np.empty_like(target_data, dtype=np.float32)
        for band in range(target_data.shape[0]):
            adjusted_data[band] = ((target_data[band] - target_means[band]) / target_stds[band]) \
                                  * ref_stds[band] + ref_means[band]

        # Ensure that the values are between [0, 255].
        adjusted_data = np.clip(adjusted_data, 0, 255).astype(np.uint8)

        # Save the output image.
        profile = target.profile
        with rasterio.open(output_path, 'w', **profile) as dst:
            dst.write(adjusted_data)
```

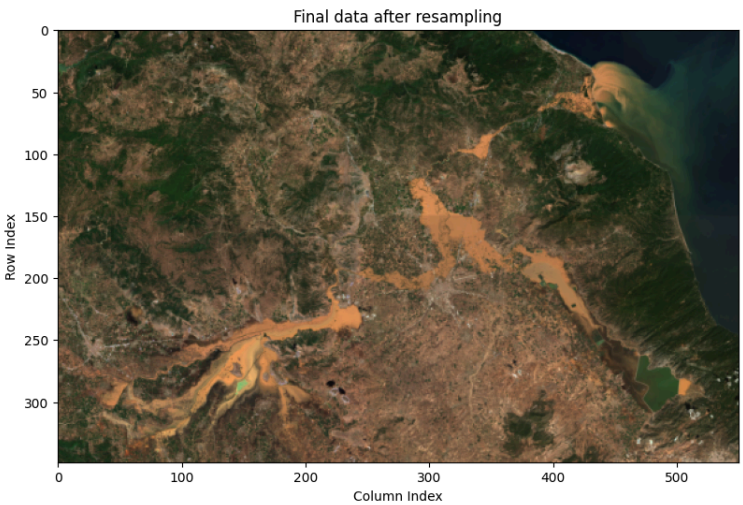
By using this code the plot will be visible without this blue shaded area.

```
# Create the reference_colors.
mosaic_rasters("reference_colors.jp2", "Greece_adjusted.jp2",)
# Adjust the piece of the bottom right.
adjust_colors("Greece_adjusted1.jp2", "reference_colors.jp2", "Greece_corrected1.jp2")

# Make the new image.
final_before = mosaic_rasters("Greece_mosaic_before.jp2", "Greece_adjusted.jp2",
                              "Greece_corrected1.jp2", "Greece_adjusted2.jp2",
                              "Greece_adjusted3.jp2", "Greece_adjusted4.jp2", "Greece_adjusted5.jp2",)
# Plot the new image.
plot_raster(final_before, 'All the pieces together')
```



Lastly the data of this picture will be resampled.

```
_, final_adjusted= resample("Greece_mosaic_before.jp2", Resampling.average, 'final_adjusted.jp2', 10, False, 0, 0, 0, 0, False, 254)
plot_raster(final_adjusted, 'Final data after resampling')
```



Testing different classifier algorithms

The code down below has been given to us. This code creates a variable with the red, green and blue pixel values for different points in the dataset. To do this, an y-variable has been created with almost 300 points, divided in flooded and non flooded points. These values will be used later on by the algorithms to determine whether the point is flooded or not.

```
y_var = gpd.read_file("Flooded_points.gpkg")

def create_x(gdp, input_file):
    # Create an empty array to store the sampled values.
    sampled_values = np.zeros((len(gdp), 3), dtype=np.uint8)

    with rasterio.open(input_file) as src:
        # Extract raster values for each point in the ground truth shapefile.
        coords = [(x, y) for x, y in zip(gdp.geometry.x, gdp.geometry.y)]
        # For every point, sample from the raster image and store the values
        # in the array.
        for idx, value in enumerate(src.sample(coords)):
            sampled_values[idx] = value

    return sampled_values

X = create_x(y_var, 'final_adjusted.jp2')
y = np.array(y_var.iloc[:,0])
```

In the code down below four machine learning algorithms will be tested on there performance to classify the data in flooded and non flooded points. This will be done by first traning the algorithms on some training dataset, afterwhich they are used on the testing dataset. Furthermore, the f1 score gives an indication on the performance of these models.

First, I'm going to create different datasets to train and test the algorithms.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42)
```

Then, I'm using this datasets to test the different algorithms.

Random Forest

```
rf = RandomForestClassifier()
# Train the algorithm.
rf.fit(X_train, y_train)
# Test the algorithm.
rf_y_pred = rf.predict(X_test)
print(f1_score(y_test, rf_y_pred, average='binary'))
```

```
0.943089430894309
```

Logistic Regression

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
log_reg_y_pred = log_reg.predict(X_test)
print(f1_score(y_test, log_reg_y_pred, average='binary'))
```

```
0.9354838709677419
```

Support Vector Classifier

```
svc = SVC()
svc.fit(X_train, y_train)
svc_y_pred = svc.predict(X_test)
print(f1_score(y_test, svc_y_pred, average='binary'))
```

```
0.9193548387096774
```

MLP Classifier

```
mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300, random_state=42)
mlp.fit(X_train, y_train)
mlp_y_pred = mlp.predict(X_test)
print(f1_score(y_test, mlp_y_pred, average='binary'))
```

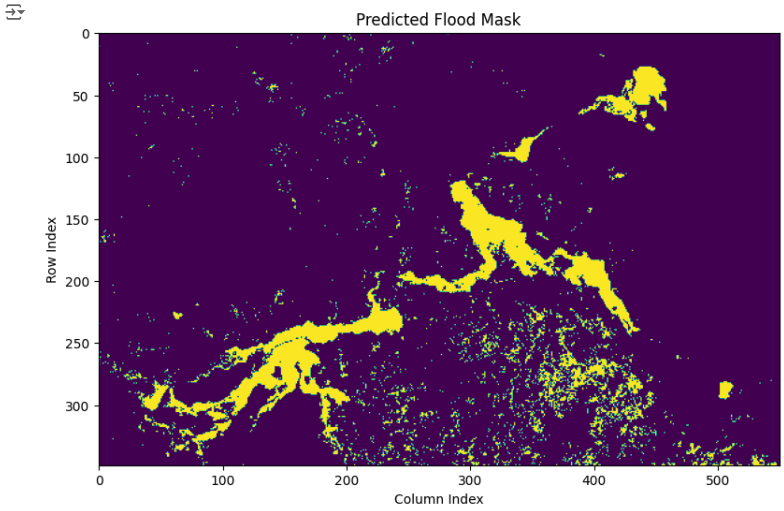
```
0.9193548387096774
```

We can see that the Random Forest algorithm performs best based on the f1 score. Therefore, this test will be used later on to predict based on the pixel color values wether a point is flooded or not.

Predicting flood

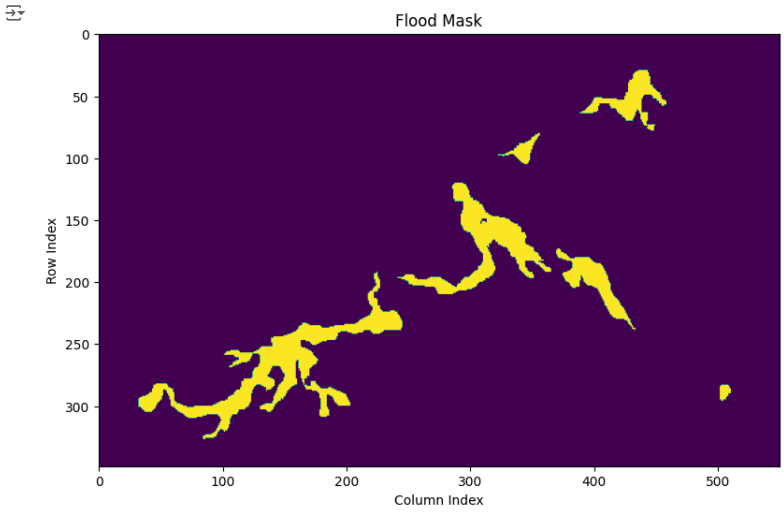
In the codes down below we will try to recreate a spatial map of the flooded area in Figure 1. The Random Forest algorithm will determine wich pixels are in flooded area based on the color values of the pixels. The resulting spatial map has been plotted thereafter.

```
# Making the predictions.
final_resh = final_adjusted.reshape(-1, 3)
final_pred = rf.predict(final_resh)

# Adding the predictions to the final_adjusted.
final_pred = final_pred.reshape(final_adjusted.shape[0],
                                final_adjusted.shape[1], 1)

final_adjusted = np.concatenate((final_adjusted, final_pred), axis=2)

# Making a plot based on the new column.
plot_raster(final_adjusted[:, :, 3], 'Predicted Flood Mask')
```



To test whether this spatial map is in line with reality, I've created a masked raster that should give an indication of what the result should have been. Furthermore, I've created a function that opens this file and returns a correspanding raster.

```
polygons = gpd.read_file("Ground_flood.gpkg")

def masked_raster(mask_data, input_file):
    # Open the file.
    with rasterio.open(input_file) as src:
        transform = src.transform
        out_shape = (src.height, src.width)
        crs = src.crs
    # Transform the file into a raster.
    rasterized_mask = rasterize(
        [(geom, 1) for geom in mask_data.geometry],
        out_shape=out_shape,
        transform=transform,
        fill=0,
        dtype='uint8')
    return rasterized_mask

# Plot the raster.
mask = masked_raster(polygons, 'final_adjusted.jp2')
plot_raster(mask, 'Flood Mask')
```



In the code down below I'm making a comparison between the predicted spatial map and what it should have been by calculating the f1, precision and recall score.

```
final_pred = final_pred.reshape(mask.shape[0], mask.shape[1])
mask = mask.reshape(-1)
final_pred = final_pred.reshape(-1)

print(precision_score(mask, final_pred, average='binary'))
print(recall_score(mask, final_pred, average='binary'))
print(f1_score(mask, final_pred, average='binary'))
```

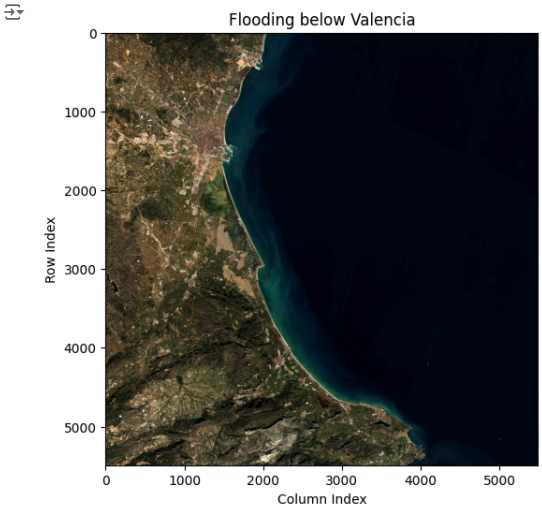
```
0.5034192209060936
0.8641651818467666
0.636212816852502
```

Application of the classifier to a different setting.

In October 2024 there have been a flood in the area slightly below Valencia. The following can be used to determine whether the Random Forest classifier can be used in a other setting without letting it adjust to the new image (no re-training).

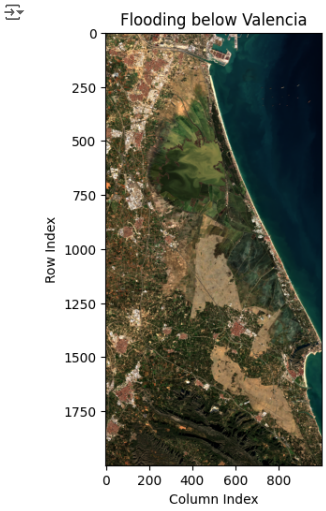
First, I'm gonna load the figure. Valencia can be seen approximatally at index [1500, 1500].

```
Valencia_before, _ = resample('Valencia.jp2', Resampling.average, 'Valencia_adjusted.jp2', 1, False, 0, 0, 0, 0, False, 0)
plot_raster(Valencia_before, 'Flooding below Valencia')
```



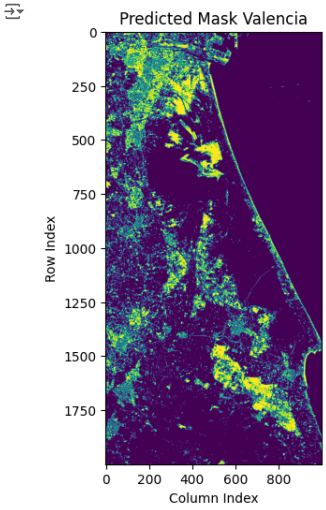
To test the classifier, I will focus on the area below Valencia.

```
Valencia_before, Valencia_adjusted = resample('Valencia.jp2', Resampling.average, 'Valencia_adjusted.jp2', 1, True, 1000, 1500, 1000, 2000, False, 254)
plot_raster(Valencia_before, 'Flooding below Valencia')
```



I'm gonna use the classifier to predict the flooded area.

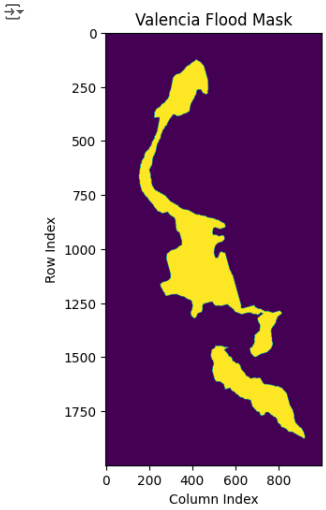
```
# Adjust the data to make it suitable for the Random Forest algorithm.
Valencia_resh = Valencia_adjusted.reshape(-1, 3)
# Predict the flooded area.
Valencia_pred = rf.predict(Valencia_resh)
# Transform the flooded area and add it to Valencia_adjusted.
Valencia_pred = Valencia_pred.reshape(Valencia_adjusted.shape[0],
                                      Valencia_adjusted.shape[1], 1)
Valencia_adjusted = np.concatenate((Valencia_adjusted, Valencia_pred), axis=2)
# Plot the spatial map.
plot_raster(Valencia_adjusted[:, :, 3], 'Predicted Mask Valencia')
```



I've again created an area that indicates the real flooded area. This will be used to compare with the predicted flooded area.

```
# I'm using the codes from before with other input values.
val_polygons = gpd.read_file("Valencia_Flooded.gpkg")
val_mask = masked_raster(val_polygons, 'Valencia_adjusted.jp2')
```

```
plot_raster(val_mask, 'Valencia Flood Mask')
```



Lastly, I'm again gonna calculate the f1, precision and recall score of this classifier.

```
Valencia_pred = Valencia_pred.reshape(val_mask.shape[0], val_mask.shape[1])
val_mask = val_mask.reshape(-1)

Valencia_pred = Valencia_pred.reshape(-1)
print(precision_score(val_mask, Valencia_pred, average='binary'))
print(recall_score(val_mask, Valencia_pred, average='binary'))
print(f1_score(mask, final_pred, average='binary'))
```

```
0.31344046229170086
0.42176541918600535
0.636212816852502
```

What stands out explicitily is that the Random Forest (without re-training) expected the red roofs of Valencia to be flooded area, however this was not the case. This is, among other miscalculations, the reason for a lowe value on the precision, recall, f1 score.