

Notebook

October 16, 2024

1 Preparation

1.1 Import library and set directory

```
[1]: import os
import xarray as xr
import geopandas as gpd

# Add the path to the designated folder containing custom modules.
import sys
sys.path.append('../src')

# Import the custom module for benthic habitat mapping.
# This module contains functions and utilities for tasks such as
# correction and classification of benthic habitats.
import benthic_mapping as bm

import matplotlib.pyplot as plt
# plt.style.use('dark_background')

from datetime import datetime

[2]: def construct_file_path(out_dir, user_year):
    # Ensure the year is a string
    user_year = str(user_year)
    # Construct the directory path
    year_folder = os.path.join(out_dir, 'atmospheric_correction', user_year)
    # Iterate through the files in the year folder
    for filename in os.listdir(year_folder):
        # Check if the file contains 'L2R' in its name
        if 'L2R' in filename:
            # Construct the full file path
            file_data_path = os.path.join(year_folder, filename)
            return file_data_path
    # If no file is found, return None
    return
```

```

# Define folder path
base_dir = os.path.abspath(os.path.join(os.getcwd(), '../..'))
data_dir = os.path.join(base_dir, 'data')
out_dir = os.path.join(base_dir, 'out')
raw_data_dir = os.path.join(data_dir, 'raw')

# Define file paths
user_year = "2018"
file_data_path = construct_file_path(out_dir, user_year)
shapefile_path_deepWater = os.path.join(out_dir, 'geom_def', 'geom_deepWater.
↳shp')
shapefile_path_sandObject = os.path.join(out_dir, 'geom_def', 'geom_sandObject.
↳shp')
shapefile_path_training_1 = os.path.join(out_dir, 'geom_def', 'geom_landWater.
↳shp')
shapefile_path_training_2 = os.path.join(out_dir, 'geom_def', '
↳geom_benthicObject.shp')

# Extract the base filename
base_filename = os.path.splitext(os.path.basename(file_data_path))[0]
base_filename = base_filename.rsplit('_', 1)[0]

```

1.2 Pre-processing dataset

1.2.1 Open and prepare the dataset

```

[3]: # Open dataset
data = xr.open_dataset(file_data_path)

# Determine if the dataset is from S2A or S2B based on the filename
if 'S2A' in file_data_path:
    print('Detected dataset from Sentinel-2A')
    variables_to_keep = {
        'transverse_mercator': 'transverse_mercator',
        'lat': 'lat',
        'lon': 'lon',
        'rhos_492': 'blue',
        'rhos_560': 'green',
        'rhos_665': 'red',
        'rhos_704': 'red_edge',
        'rhos_833': 'nir',
        'rhos_1614': 'swir1',
        'rhos_2202': 'swir2'
    }
elif 'S2B' in file_data_path:
    print('Detected dataset from Sentinel-2B')

```

```

variables_to_keep = {
    'transverse_mercator': 'transverse_mercator',
    'lat': 'lat',
    'lon': 'lon',
    'rhos_492': 'blue',
    'rhos_559': 'green',
    'rhos_665': 'red',
    'rhos_704': 'red_edge',
    'rhos_833': 'nir',
    'rhos_1610': 'swir1',
    'rhos_2186': 'swir2'
}
else:
    raise ValueError("The dataset file path does not indicate whether it is S2A_
↳ or S2B.")

# Create the new dataset
new_vars = {}
for old_name, new_name in variables_to_keep.items():
    if old_name in data:
        # Select the variable and transpose if needed
        variable = data[old_name]
        new_vars[new_name] = variable

# Construct the new dataset
ds = xr.Dataset(new_vars)

# Preserve selected attributes
attributes_to_keep = [
    'generated_by', 'generated_on', 'contact', 'product_type',
↳ 'metadata_profile', 'Conventions',
    'sensor', 'isodate', 'global_dims', 'sza', 'vza', 'raa', 'scene_xrange',
↳ 'scene_yrange',
    'scene_dims', 'scene_pixel_size', 'data_dimensions', 'data_elements',
↳ 'acolate_version',
    'acolate_file_type', 'tile_code', 'proj4_string', 'pixel_size', 'uoz',
↳ 'uwv', 'wind',
    'pressure', 'oname'
]
ds.attrs = {key: data.attrs[key] for key in attributes_to_keep}

# Close the original dataset
data.close()

```

Detected dataset from Sentinel-2A

1.2.2 Reset encoding and define projection

```
[4]: # Reset encoding
ds = ds.drop_encoding()

# Set CRS
wkt = ds.attrs['proj4_string']
ds = ds.rio.write_crs(wkt, inplace=True)

# Drop 'grid_mapping'
for var in ds.data_vars:
    if 'grid_mapping' in ds[var].attrs:
        del ds[var].attrs['grid_mapping']

# Print the current CRS
print("Current CRS:", ds.rio.crs)
```

Current CRS: EPSG:32748

2 Image Processing

2.1 Sun Glint Correction (Hedley et al., 2005)

```
[5]: # Subsetting sample area for the Sun Glint Correction
# Read shapefile and desired year for the input
var_select = ['blue', 'green', 'red', 'red_edge', 'nir'] # Variables to select
# from the dataset
gdf = gpd.read_file(shapefile_path_deepWater) # Load shapefile containing the
# region of interest
desired_year = int(user_year) # Convert user_year to integer

# Mask the dataset based on the shapefile and desired year
samples = bm.mask_dataset(
    ds[var_select], gdf, desired_year
)

# Compute sun glint correction using the 'sunglint_correction' function from
# the module
# Note: The 'vars_ignore' parameter excludes 'lat' and 'lon' from the
# correction process. Default set to None
sg_ds = bm.sunglint_correction(ds, samples, 'nir', vars_ignore=['lat', 'lon'])
```

Minimum NIR brightness (MinNir): 0.0013025731313973665

Regression results for blue: slope=0.697225323118322,

r_value=0.8061301117812573, p_value=0.0

Regression results for green: slope=0.7944032714417325,

r_value=0.8642988664592968, p_value=0.0

Regression results for red: slope=0.6077591929385986,

```
r_value=0.6184170818185591, p_value=0.0
Regression results for red_edge: slope=0.24139304599378733,
r_value=0.39967829489329265, p_value=0.0
```

Slope information not found for variable 'swir1'. Skipping correction.
Slope information not found for variable 'swir2'. Skipping correction.

2.2 Depth Invariant Index (Green et al., 2000)

```
[6]: # Subsetting sample area for the DII calculation
# Read the shapefile containing the region of interest
gdf = gpd.read_file(shapefile_path_sandObject) # Load shapefile for sand
      ↪ object classification

# Mask the dataset based on the shapefile and desired year
samples = bm.mask_dataset(
    sg_ds, gdf, desired_year
)

# Define pairs of bands for which to calculate k-ratio and Depth Invariant
      ↪ Index (DII)
band_pairs = [
    ('blue_sg', 'green_sg'),
    ('blue_sg', 'red_sg'),
    ('blue_sg', 'red_edge_sg'),
    ('green_sg', 'red_sg'),
    ('green_sg', 'red_edge_sg'),
    ('red_sg', 'red_edge_sg')
]

# Calculate the water column corrected dataset using the specified band pairs
wc_ds = bm.water_column_correction(sg_ds, samples, band_pairs)
```

Calculating DII for bands blue_sg and green_sg with k-ratio: 0.7502886250912342

Calculating DII for bands blue_sg and red_sg with k-ratio: 0.4127506511334793

Calculating DII for bands blue_sg and red_edge_sg with k-ratio:
0.020825071212762225

Calculating DII for bands green_sg and red_sg with k-ratio: 0.7893887107338351

Calculating DII for bands green_sg and red_edge_sg with k-ratio:
0.3675400637806532

Calculating DII for bands red_sg and red_edge_sg with k-ratio:
0.8053349503971705

2.3 Spectral Indices

2.3.1 Normalized Difference

```
[7]: # Define pairs of variables for which to compute normalized difference indices
variable_pairs = [
    ('nir', 'red'),
    ('green', 'swir1'),
    ('nir', 'green'),
    ('green', 'red'),
    ('green', 'red_edge')
]

# Define names for the resulting normalized difference indices
var_names = ['ndvi', 'mndwi', 'gndvi', 'ngrdi_red', 'ngrdi_red_edge']

# Compute the normalized difference indices
si_ds = bm.normalized_difference(ds, variable_pairs, var_names)
```

2.3.2 Non-Normalized Difference

```
[8]: # Calculate EVI
evi = (ds['nir'] - ds['red']) / (ds['nir'] + 6 * ds['red'] - 7.5 * ds['blue'] + 1)

# Calculate AWEI
awei = 4 * (ds['green'] - ds['swir2']) - (0.25 * ds['nir'] + 2.75 * ds['swir1'])

# Create DataArray for EVI with attributes
si_ds['evi'] = xr.DataArray(
    data=evi,
    dims=ds['nir'].dims,
    coords=ds['nir'].coords,
    name='evi',
    attrs={
        'long_name': 'Enhanced Vegetation Index (EVI)',
        'formula': '(NIR - RED) / (NIR + 6 * RED - 7.5 * BLUE + 1)',
        'units': '1',
        'date_created': datetime.utcnow().isoformat(),
    }
)

# Create DataArray for AWEI with attributes
si_ds['awei'] = xr.DataArray(
    data=awei,
    dims=ds['green'].dims,
    coords=ds['green'].coords,
    name='awei',
```

```

    attrs={
        'long_name': 'Automated Water Extraction Index (AWEI)',
        'formula': '4 * (GREEN - SWIR2) - (0.25 * NIR + 2.75 * SWIR1)',
        'units': '1',
        'date_created': datetime.utcnow().isoformat(),
    }
)

```

2.4 Merge Processed Dataset

```

[9]: # Merge dataset
     clf_ds = xr.merge([ds, sg_ds, wc_ds, si_ds])

```

3 Classification

```

[10]: import pandas as pd
       import numpy as np

       from sklearn.ensemble import RandomForestClassifier
       from sklearn.model_selection import train_test_split
       from sklearn.metrics import classification_report, confusion_matrix, \
           accuracy_score, cohen_kappa_score

```

3.1 Land-Water Classification

```

[11]: # Subsetting sample area for the classification
       # Read the shapefile containing the region of interest
       gdf = gpd.read_file(shapefile_path_training_1)

       # Extract labeled samples for the specified year
       samples = bm.labeled_samples(
           clf_ds, gdf, 'class', desired_year
       )

       # Define the list of features to be used in the classification model
       features = [
           'blue', 'green', 'red', 'red_edge', 'nir', 'swir1',
           'blue_sg', 'green_sg', 'red_sg', 'red_edge_sg',
           'dii_blue_sg_green_sg', 'dii_blue_sg_red_sg',
           'dii_blue_sg_red_edge_sg', 'dii_green_sg_red_sg',
           'dii_green_sg_red_edge_sg', 'dii_red_sg_red_edge_sg',
           'mndwi', 'awei',
       ]

       # Prepare the feature matrix (X) and target labels (y) from the given samples
       X, y = bm.prepare_samples(samples, features, 'label')

```

```
[12]: # Classification procedure
# Split the samples into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Initialiaze a Random Forest classifier
clf = RandomForestClassifier(
    n_estimators=80, random_state=42, max_depth=15,
    min_samples_leaf=1, min_samples_split=2
)

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Retrieve the features importance
importances = clf.feature_importances_
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': importances
})
importance_df = importance_df.sort_values(
    by='Importance', ascending=False
).reset_index(drop=True)

# Print feature importances
importance_df

# Save the DataFrame to a CSV file
export_file_path = os.path.join(out_dir, 'csv',
    f"{'feature_importances_classification_1'}_{user_year}.csv")
importance_df.to_csv(export_file_path)
```

```
[13]: # Accuracy assessment
# Predict the labels for the test set using the trained classifier
y_pred = clf.predict(X_test)

# Print the reports
print(classification_report(y_test, y_pred))
print(f"{confusion_matrix(y_test, y_pred)}\n")
print(f"accuracy score:{accuracy_score(y_test, y_pred)}")
print(f"cohen's kappa:{cohen_kappa_score(y_test, y_pred)}")
```

	precision	recall	f1-score	support
1	1.00	1.00	1.00	157530
2	1.00	1.00	1.00	1527
3	1.00	0.99	1.00	1294

accuracy			1.00	160351
macro avg	1.00	1.00	1.00	160351
weighted avg	1.00	1.00	1.00	160351

```
[[157530    0    0]
 [    1  1524    2]
 [    5    3 1286]]
```

accuracy score:0.9999314004901747

cohen's kappa:0.9980221310476876

```
[14]: # Predict full set and reshape into the original spatial dimensions
# Get the dimensions (height and width) of the dataset
height = clf_ds.sizes['y']
width = clf_ds.sizes['x']

# Stack the feature data into a 2D array 'X_full'
# Create a mask to identify rows with valid (non-NaN) feature values
# Predict the labels for the entire dataset using the trained RandomForest model
X_full = np.stack([clf_ds[var].values.flatten() for var in features], axis=1)
mask = ~np.isnan(X_full).any(axis=1)
y_full_pred = clf.predict(X_full)

# Initialize an array filled with NaN values to store reshaped predictions
y_full_pred_reshaped = np.full((height, width), np.nan)
# Flatten the predicted labels and create a flattened array filled with NaN
# values
y_full_pred_flattened = y_full_pred.reshape(-1)
y_full_pred_reshaped_flat = np.full(height * width, np.nan)
# Apply the mask to place the predicted labels into the correct positions in
# the flattened array
y_full_pred_reshaped_flat[mask] = y_full_pred_flattened[mask]
# Reshape the flattened array back into the original 2D shape (height, width)
y_full_pred_reshaped = y_full_pred_reshaped_flat.reshape((height, width))

# Create a new xarray.DataArray to store the predictions in the dataset
clf_ds['predictions'] = xr.DataArray(
    y_full_pred_reshaped,
    dims=clf_ds[list(clf_ds.data_vars)[0]].dims,
    coords=clf_ds[list(clf_ds.data_vars)[0]].coords,
    name='predictions',
    attrs={
        'long_name': 'Predicted label',
        'model': 'RandomForestClassifier',
        'data_created': datetime.utcnow().isoformat(),
    }
)
```

```
)
```

```
[15]: # Store the data for export
      clf_ds_export_1 = clf_ds
```

```
[16]: # Create a boolean mask where the predictions are equal to 2 (shallow water)
      pred_equals_2 = clf_ds['predictions'] == 2

      # Filter the dataset variables to include only those where the predictions
      # equal 2
      # This retains only the values for which the condition (predictions == 2) is
      # true
      filtered_ds = xr.Dataset({var_name: clf_ds[var_name].where(pred_equals_2) for
      # var_name in clf_ds.data_vars})

      # Copy attributes from the original dataset to the new filtered dataset
      for attr in clf_ds.attrs:
          filtered_ds.attrs[attr] = clf_ds.attrs[attr]

      # Copy coordinates from the original dataset to the new filtered dataset
      for coord in clf_ds.coords:
          filtered_ds[coord] = clf_ds[coord]

      # Update the original dataset to the new filtered dataset
      clf_ds = filtered_ds
```

```
[17]: # Create a boolean mask where the predictions are equal to 2 (shallow water)
      # pred_equals_2 = clf_ds['predictions'] == 2
      pred_equals_2 = clf_ds_export_1['predictions'] == 2

      # Filter the dataset variables to include only those where the predictions
      # equal 2
      # This retains only the values for which the condition (predictions == 2) is
      # true
      filtered_ds = xr.Dataset({var_name: clf_ds_export_1[var_name].
      # where(pred_equals_2) for var_name in clf_ds_export_1.data_vars})

      # Copy attributes from the original dataset to the new filtered dataset
      for attr in clf_ds_export_1.attrs:
          filtered_ds.attrs[attr] = clf_ds_export_1.attrs[attr]

      # Copy coordinates from the original dataset to the new filtered dataset
      for coord in clf_ds_export_1.coords:
          filtered_ds[coord] = clf_ds_export_1[coord]

      # Update the original dataset to the new filtered dataset
      clf_ds = filtered_ds
```

3.2 Benthic Classification

```
[18]: # Subsetting sample area for the classification
# Read the shapefile containing the region of interest
gdf = gpd.read_file(shapefile_path_training_2)

# Extract labeled samples for the specified year
samples = bm.labeled_samples(
    clf_ds,gdf, 'class', desired_year
)

# Define the list of features to be used in the classification model
features = [
    'blue', 'green', 'red', 'red_edge', 'nir', 'swir1',
    'swir2', 'blue_sg', 'green_sg', 'red_sg', 'red_edge_sg',
    'dii_blue_sg_green_sg', 'dii_blue_sg_red_sg',
    'dii_blue_sg_red_edge_sg', 'dii_green_sg_red_sg',
    'dii_green_sg_red_edge_sg', 'dii_red_sg_red_edge_sg',
    'ndvi', 'mndwi', 'gndvi', 'ngrdi_red', 'ngrdi_red_edge',
    'evi', 'awei',
]
# Prepare the feature matrix (X) and target labels (y) from the given samples
X, y = bm.prepare_samples(samples, features, 'label')
```

```
[19]: # Classification procedure
# Split the samples into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Initialize a Random Forest classifier
clf = RandomForestClassifier(
    n_estimators=200, random_state=42, max_depth=15,
    min_samples_leaf=1, min_samples_split=2
)

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Retrieve the features importance
importances = clf.feature_importances_
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': importances
})
importance_df = importance_df.sort_values(
    by='Importance', ascending=False
).reset_index(drop=True)
```

```
# Print feature importances
importance_df

# Save the DataFrame to a CSV file
export_file_path = os.path.join(out_dir, 'csv',
    ↳f"{'feature_importances_classification_2'}_{user_year}.csv")
importance_df.to_csv(export_file_path)
```

```
[20]: # Accuracy assessment
# Predict the labels for the test set using the trained classifier
y_pred = clf.predict(X_test)

# Print the reports
print(classification_report(y_test, y_pred))
print(f"{'confusion_matrix(y_test, y_pred)}\n")
print(f"{'accuracy score:{accuracy_score(y_test, y_pred)}"}")
print(f"{'cohen's kappa:{cohen_kappa_score(y_test, y_pred)}"}")
```

	precision	recall	f1-score	support
1	0.83	0.69	0.75	144
2	0.95	0.99	0.97	343
3	0.91	1.00	0.95	323
4	0.97	0.93	0.95	490
accuracy			0.94	1300
macro avg	0.91	0.90	0.91	1300
weighted avg	0.93	0.94	0.93	1300

```
[[100  10  19  15]
 [  5 338   0   0]
 [  0   0 323   0]
 [ 16   7  12 455]]
```

```
accuracy score:0.9353846153846154
cohen's kappa:0.9094654231703849
```

```
[21]: # Predict full set and reshape into the original spatial dimensions
# Get the dimensions (height and width) of the dataset
height = clf_ds.sizes['y']
width = clf_ds.sizes['x']

# Stack the feature data into a 2D array 'X_full'
# Create a mask to identify rows with valid (non-NaN) feature values
# Predict the labels for the entire dataset using the trained RandomForest model
X_full = np.stack([clf_ds[var].values.flatten() for var in features], axis=1)
```

```

mask = ~np.isnan(X_full).any(axis=1)
y_full_pred = clf.predict(X_full)

# Initialize an array filled with NaN values to store reshaped predictions
y_full_pred_reshaped = np.full((height, width), np.nan)
# Flatten the predicted labels and create a flattened array filled with NaN
↳ values
y_full_pred_flattened = y_full_pred.reshape(-1)
y_full_pred_reshaped_flat = np.full(height * width, np.nan)
# Apply the mask to place the predicted labels into the correct positions in
↳ the flattened array
y_full_pred_reshaped_flat[mask] = y_full_pred_flattened[mask]
# Reshape the flattened array back into the original 2D shape (height, width)
y_full_pred_reshaped = y_full_pred_reshaped_flat.reshape((height, width))

# Create a new xarray.DataArray to store the predictions in the dataset
clf_ds['predictions'] = xr.DataArray(
    y_full_pred_reshaped,
    dims=clf_ds[list(clf_ds.data_vars)[0]].dims,
    coords=clf_ds[list(clf_ds.data_vars)[0]].coords,
    name='predictions',
    attrs={
        'long_name': 'Predicted label',
        'model': 'RandomForestClassifier',
        'data_created': datetime.utcnow().isoformat(),
    }
)

```

4 Export dataset

```

[22]: def add_time_coordinate(dataset, base_filename):

    # Split the base filename to extract date and time components
    parts = base_filename.split('_')

    # Extract year, month, day, hour, and minute from the filename
    year = int(parts[2])
    month = int(parts[3])
    day = int(parts[4])
    hour = int(parts[5])
    minute = int(parts[6])

    # Create a datetime object for the time coordinate
    time_coord = [datetime(year, month, day, hour, minute)]

    # Assign the time coordinate to the dataset's coordinates

```

```

dataset = dataset.assign_coords(time=('time', time_coord))

# Set attributes for the 'time' coordinate
dataset['time'].attrs = {
    'standard_name': 'time',
    'long_name': 'time',
}

# Expand dimensions for all data variables to include the 'time' dimension
for var in dataset.data_vars:
    dataset[var] = dataset[var].expand_dims('time', axis=0)

return dataset

```

```

[23]: def generate_encoding(dataset):
    encoding = {}

    for var_name in dataset.data_vars:
        var = dataset[var_name]
        encoding[var_name] = {
            "dtype": "float32",
            'zlib': False,
            'shuffle': False,
            'complevel': 0,
            'fletcher32': False,
            'contiguous': True,
            'chunksizes': None,
            'original_shape': (2343, 2530),
            'grid_mapping': 'transverse_mercator'
        }

    return encoding

```

```

[24]: # Save the data to a new NetCDF file
encoding = generate_encoding(clf_ds_export_1)
clf_ds_export_1 = add_time_coordinate(clf_ds_export_1, base_filename)
export_file_path = os.path.join(out_dir, 'processed', f"{base_filename}_{'L2F'}.
    ↪nc")
clf_ds_export_1.to_netcdf(export_file_path, encoding=encoding)

```

```

[25]: # Save the data to a new NetCDF file
encoding = generate_encoding(clf_ds)
clf_ds = add_time_coordinate(clf_ds, base_filename)
export_file_path = os.path.join(out_dir, 'classification',
    ↪f"{base_filename}_{'L3W'}.nc")
clf_ds.to_netcdf(export_file_path, encoding=encoding )

```