Eo-learn core features

The core library's structure consists of the four main building blocks: EOPatch, EOTask, EOWorkflow and EOExecutor [48,49].

EOPatches (Figure S1) are class-objects that can store various features, according to FeatureTypes (eo-learn, 2018), under a common bounding box. Multi-temporal and multi-band imagery in EOPatches are multi-dimensional NumPy arrays [91] where the dimensions vary, but usually express raster pixel width and height, image ingestion time and bands. EOPatches control the data format for a given FeatureType automatically. They can serve to split (to patch) an area of interest we want to classify, which allows to scale the training and classification process and thus reduce computational requirements and the amount of data needed. It is therefore possible to prepare data, train an estimator, classify imagery and verify the results within a set of patches, which can be stored as *.npy files [92]. As utilized in this experiment, EOPatches contained geometries for the study area, training/testing data and multi-temporal Sentinel-2 imagery.

```
EOPatch(
  data: {'BANDS': NumPy.array} # shape = time×width×height×bands
        {'SPECTRAL_INDEX_NDVI': NumPy.array}
  mask: {'SCL_MASK': NumPy.array} # shape = time×width×height×bands
  mask_timeless: {'LULC': NumPy.array} # shape = width×height×bands
  meta_info: {} # anything
  bbox: SentinelHub.Bbox # Sentinel Hub bounding box object
  timestamp: [datetime.date(2019,9,1), ...] # list of datetime.date objects
)
```

Figure S1. A sample EOPatch object with some of its FeatureTypes (i.e. data, mask etc.) and the required value after the colon (i.e. Python dictionaries with multi-dimensional NumPy arrays).

EOTask (Figure S2) is an operational class-object that can execute methods on EOPatches. There are numerous native operations in eo-learn to work with satellite data, inherited from the design of EOTask (i.e. adding features to EOPatches or exporting results to various formats). Custom EOTasks can be created with their own attributes and methods. Nevertheless, they always have to implement the execute method, which performs the desired operation and returns an altered EOPatch. EOTasks used in this experiment are further described accordingly.

```python
class DerivateProduct(EOTask):  # Inherit from eo-learn's EOTask class
    """
    Custom EOTask that calculates a user-chosen derivate product (index or
similar).
    """
    def __init__(self, derivate_name, equation):
        self.derivate_name = derivate_name
        self.equation = equation

    def arbitrary_method(self, equation):
       # Do something useful
       return result

    def execute(self, eopatch, **keyword_arguments):
        # Mandatory method that orchestrates operations of other methods if
           applicable
        # Keyword arguments can be supplied in EOWorkflow
        derivate_product = self.arbitrary_method(self.equation)
        # Load or alter data to EOPatch
        eopatch.data[self.derivate_name] = derivate_product

        return eopatch # Returns eopatch. In EOWorkflows EOPatch is passed onto
                         another EOTask
```

Figure S2. A sample EOTask that calculates a multi-image feature, such as normalized difference indices. Its components are explained in Python comments.

```python
workflow = LinearWorkflow( # Instantiating EOWorkflow (can be other than
                             linear)
    load_eopatches, # Instance of some eo-learn's native EOTask
    derivate_product, # Instance of DerivateProduct EOTask
    save_eopatches
)

external_args = [{ # On-the-fly arguments, such as which EOPatch to load
        load: {'eopatch_folder': f'eopatch{alternating_id}'},
        save: {'eopatch_folder': f'new_eopatch{alternating_id}'}
    }]

executor = EOExecutor(workflow, # Instantiating EOExecutor
                      external_args
                  )
executor.run(workers=1) # specify if multiprocess on more CPU threads and
                          run workflow
executor.make_report() # Make report
```

Figure S3. A sample EOWorkflow and EOExecutor as used to pipeline EOTasks. Components are explained in Python comments.

EOWorkflow (Figure S3) can be likened to a flowchart diagram or a model builder in common GIS applications. It is a class-object that defines EOTask execution succession, which can be linear or non-linear. In EOWorkflow, EOPatches share data for the underlying analyses and among each other, coordinated by EOTasks. External arguments for executing EOTasks can be defined. EOWorkflows in this experiment encompass filling EOPatches, preparing data for classification and predicting results. EOExecutor is a class-object that executes the whole workflow, enabling to parallelize executions (if performed on multiple EOPatches). It outputs a report (log) of how the execution was performed. EOExecutors were used along with the EOWorkflows.

```python
import sys
import os
import datetime
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
from shapely.geometry import Polygon, box
from sentinelhub import BBoxSplitter, BBox, CRS


class Project:
    '''
    An auxiliary class to create and manage the pipeline's data in the project folder.
    '''

    def __init__(self, path_to_project):
        self.FOLDER = self.set_FOLDER(path_to_project)

    def set_FOLDER(self, path):
        '''
            Prompts to set a project folder or create a new one.
            Should be set at any Jupyter Notebook which is an instance of the pipeline project.
            '''
        if Project.is_project(path):
            decision = input(f'Project {path} already exists, do you want to use it (1) or create a new one
(2)? ')
            if int(decision) == 1:
                print(f'Project {path} has been set to be used.')
                return Project.get_existing_project(path)
            elif int(decision) == 2:
                print('Change the project name at project instance __init__.')
                return None
        os.mkdir(path)
        print(f'Project created at: {path}')
        return path

    @staticmethod
    def is_project(path):
        '''
            Checks if project exists.
            '''
        return any(folder == path for folder in os.listdir('.'))

    @staticmethod
    def get_existing_project(path):
        '''
            Gets the existing project path, if it exists.
            '''
        for folder in os.listdir('.'):
            if folder == path:
                return folder

    def __str__(self):
        return f'Project used: {self.FOLDER}'
```

```
class AOI:
    '''
    A class to set the area of interest and prepare some essential variables for the Patcher class.
    The area of interest must be in the ESRI Shapefile or GeoJSON formats.
    '''

    def __init__(self, path_to_aoi, crs=CRS.UTM_33N):
        self.crs = crs
        self.path = self.set_AOI(path_to_aoi)
        self.gdf = self.set_AOI_gdf()
        self.shape = self.get_AOI_shape()
        self.dimensions = self.get_AOI_dimensions()

    def set_AOI(self, path):
        '''
            Sets filepath to the file containing the AOI.
            '''
        while not os.path.isfile(path):
            path = input(f'Path {path} for your area of interest does not exist. Change it: ')
        return path

    def set_AOI_gdf(self):
        '''
            Loads a GeoJSON or Shapefile to a Geopandas GeoGataframe.
            '''
        return gpd.read_file(self.path)

    def get_AOI_crs(self):
        return crs

    def get_AOI_shape(self):
        '''
            Extracts shape from the geodataframe.
            '''
        return self.gdf.geometry.values[0]

    def get_AOI_dimensions(self):
        '''
            Obtains dimensions of the AOI.
            '''
        shape = self.shape
        return (shape.bounds[2] - shape.bounds[0], shape.bounds[3] - shape.bounds[1])

    def convert_desired_CRS(self):
        '''
            For safety, the AOI is converted to the UTM33N CRS. This has to be done manually now.
            The user has to know which UTM zone is their imagery in.
            '''
        self.gdf = self.gdf.to_crs(crs={'init': CRS.ogc_string(self.crs)})

    def __str__(self):
        return f'''AOI:
- path: {self.crs}
- dimensions: {self.dimensions[0]} x {self.dimensions[1]} m
- CRS: {self.crs}'''


class Patcher:
    '''
    A class that splits the AOI into patches based on the given dimensions. Creates list of bboxes
    and info list with parent bbox (bbox of the aoi) and spatial indexes of bboxes.
    '''
```

```python
    def __init__(self, project_folder, crs=CRS.UTM_33N):
        self.crs = crs
        self.project_folder = project_folder
        self.bbox_list, self.info_list = None, None
        self.xy_splitters = None
        self.gdf = None
        self.selected_patches = None
        self.patch_gdf_bboxes = None

    def get_xy_splitters(self, aoi_dimensions, patch_factor=1):
        '''
            Diminishes the dimensions of the AOI to less-than-hundred numbers to
            represent number of bounding boxes in West-East and North-South directions.
            '''
        x, y = aoi_dimensions
        while x >= 100 or y >= 100:
            x /= 10
            y /= 10
        self.xy_splitters = (int(x*patch_factor),int(y*patch_factor))
        print(f'Area will be split into {self.xy_splitters[0]} x {self.xy_splitters[1]} patches. You can adjust it by
patch_factor.')
        return self.xy_splitters

    def split_bboxes(self, shape):
        '''
            Uses Sentinel Hub BBoxSplitter to split the AOI to bounding boxes
            intersecting or being within the AOI.
            The list of bounding boxes and information about them is retrieved.
            '''
        bbox_splitter = BBoxSplitter([shape], self.crs, self.xy_splitters)
        self.bbox_list = np.array(bbox_splitter.get_bbox_list())
        self.info_list = np.array(bbox_splitter.get_info_list())

    def get_patch_gdf(self, subset_patch=None, save=False):
        '''
            It creates a GeoDataFrame from patches according to the subset.
            '''
        subset = self.select_patch_subset(subset_patch)
        geometry = [Polygon(bbox.get_polygon()) for bbox in self.bbox_list[subset]]
        idxs_x = [info['index_x'] for info in self.info_list[subset]]
        idxs_y = [info['index_y'] for info in self.info_list[subset]]
        df = pd.DataFrame({'index_x': idxs_x,
                    'index_y': idxs_y,
                    'patch_id': np.arange(0, len(geometry))
                    })
        gdf = gpd.GeoDataFrame(df,
                     crs={'init': CRS.ogc_string(self.crs)},
                     geometry=geometry)
        gdf['centre_point'] = gdf.apply(lambda row: Patcher.getXY(row.geometry.centroid), axis=1)
        self.gdf = gdf
        if save: self.save_patches_as_shp()

    def select_patch_subset(self, ID):
        '''
            Selects central patch ID and get also IDs of those patches that surround the central
patch.
            '''
        if ID is None:
            return np.array([ID for ID in range(len(self.info_list))])
        aux = [idx for idx, [bbox, info] in enumerate(zip(self.bbox_list, self.info_list))
            if (abs(info['index_x'] - self.info_list[ID]['index_x']) <= 1 and abs(info['index_y'] - self.info_list[ID]['index_y']) <= 1)]

        # Renumbering the patches
```

```python
    return np.transpose(np.fliplr(np.array(aux).reshape(3, 3))).ravel()

  @staticmethod
  def getXY(centroid):
    '''
        Auxiliary method to get the centroid of each patch for attributing and map-making.
        '''
    return (centroid.x, centroid.y)

  def get_patch_map(self, aoi, save=False):
    '''
        Auxiliary method to print an overview map of bboxes of the AOI.
        '''
    fig, ax = plt.subplots(figsize=(20, 20))
    aoi.plot(ax=ax)
    self.gdf.plot(ax=ax, facecolor='None', edgecolor='r')
    for i in range(len(self.gdf)):
      plt.annotate(s=self.gdf.patch_id[i], xy=self.gdf.centre_point[i], color='r')
    if save: plt.savefig(input())

  def save_patches_as_shp(self):
    '''
        Saves a GeoDataFrame of selected patches as an ESRI Shapefile.
        '''
    path = f'{self.project_folder}/patches{self.xy_splitters[0]}x{self.xy_splitters[1]}'
    gdf_to_save = self.gdf
    gdf_to_save = gdf_to_save.drop('centre_point', axis=1)
    if not os.path.isdir(path):
      os.mkdir(path)
    gdf_to_save.to_file(f'{path}/patches{self.xy_splitters[0]}x{self.xy_splitters[1]}.shp', driver='ESRI Shapefile')
```

Figure S4. The aoi.py module source code

```python
# Native Python libraries
import os
import datetime
from contextlib import contextmanager
import zipfile
import re

# Non-native libraries
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
from shapely.geometry import Polygon, box
from tqdm import tqdm
import gdal
from sentinelsat import SentinelAPI
import rasterio
import rasterio.warp
from rasterio import Affine, MemoryFile
from rasterio.enums import Resampling
from rasterio.plot import show
from rasterio.mask import mask

# eo-learn + sentinelhub libraries
from eolearn.core import EOTask, EOPatch, EOExecutor, LinearWorkflow, FeatureType, OverwritePermission, LoadFromDisk,
SaveToDisk, SaveTask
from eolearn.io import S2L1CWCSInput, ExportToTiff
from eolearn.mask import AddCloudMaskTask, get_s2_pixel_cloud_detector, AddValidDataMaskTask
from eolearn.geometry import VectorToRaster, PointSamplingTask, ErosionTask
from eolearn.features import LinearInterpolation, SimpleFilterTask
from sentinelhub import BBoxSplitter, BBox, CRS, CustomUrlParam


class S2L2AImages:
    '''
    A class to get Sentinel-2 L2A images.
    For now, this class can only obtain images for patches under a single Sentinel-2 scene,
    under a single UTM zone and where L2A coverage is available. The script does not mosaic
    automatically.
    '''
    def __init__(self, patch_gdf, path_to_download):
        self.patch_gdf = patch_gdf
        self.path_to_download = path_to_download
        self.bbox = None
        self.available = pd.DataFrame()
        self.downloaded = pd.DataFrame()
        self.api = None
        self.products = None

    def get_patches_bbox(self):
        '''
        Gets total bounding box of selected patches for downloading S2L2A images
        through sentinelsat API. Only adjacent or very close patches should can be chosen for now.
        '''

        self.bbox = box(*self.patch_gdf.to_crs('EPSG:4326').geometry.total_bounds)
        return self.bbox

    def get_available(self, esa_sci_hub_credentials=('username', 'password'), date_range=('20190620', '20190627'),
                cloudcov=(0, 100)):
```

```
        '''
        According to the user-specified parameters, available Sentinel-2 L2A images are obtained from Sentinelsat API.
        '''

        assert self.bbox != None, 'Bbox for patches is empty. Use self.get_patches_bbox to obtain one.'
        self.api = SentinelAPI(*esa_sci_hub_credentials)
        self.products = self.api.query(self.bbox,
                    date=date_range,
                    platformname='Sentinel-2',
                    cloudcoverpercentage=cloudcov,
                    producttype='S2MSI2A')
        assert not self.products == None, 'There were no sentinelsat api or available images found.'
        self.available = self.api.to_geodataframe(self.products).sort_values(by='ingestiondate')
        return self.available

    def select(self, selected=None):
        '''
        Selects desired images by date. For example, if we do not want the whole time series but some particular dates.
        '''

        if selected:
            condition = self.available['ingestiondate'].apply(lambda x: datetime.datetime.strftime(x, '%Y-%m-%d')).isin(selected)
            self.downloaded.append(self.available[condition]).sort_values(by='ingestiondate')
        else:
            self.downloaded = self.available
        return self.downloaded

    def download(self):
        '''
        Wrapper for downloading images.
        '''

        assert not self.available.empty, 'There were no sentinelsat api or available images found.'
        self.downloaded.apply(self.download_aux, axis=1)

    def download_aux(self, image_row):
        '''
        Auxiliary method that controls if the zip file already exists. If not, it downloads a single-image
        zipfile and moves on to the next one as this method is invoked by self.download -> DataFrame.apply.
        '''
        zip_file = image_row[0]+'.zip'
        if not os.path.isdir(zip_file):
            self.api.download(image_row[33], directory_path=self.path_to_download)



ass CustomInput(EOTask):
    '''
    A class that prepares Sentinel-2 images to further work in the pipeline.
    '''

    # Bands in the original Sentinel-2 zipfile (SAFE file) are groupped by geometric resolutions
    # Within these groups, they are not ordered by the band number
    # BAND_ORDER is a hash table for ordering the bands by band number later in the process and storing
    # the hash information to the META_DATA FeatureType
    BAND_ORDER = np.array([2,1,0,6,3,4,5,7,8,9])
    BAND_NAMES = np.array(['B4', 'B3', 'B2', 'B8', 'B5', 'B6', 'B7', 'B8A', 'B11', 'B12'])


    def __init__(self, images_gdf, path_to_images, custom_input_name):
        self.images = images_gdf
```

```python
        self.path_to_images = path_to_images
        self.custom_input_name = custom_input_name

    def execute(self, **kwargs):
        '''
        Mandatory EOTask method, orchestrator for the Sentinel-2 imagery processing, using the methods below.
        '''
        # Initializing a new EOPatch
        eopatch = EOPatch()

        # Processing S2L2A data and CLM
        band_sets = self.images.apply(lambda row: self.process_bands(*self.load_bands(row), kwargs.get('patch_bbox')), axis=1)

        # Adding new features to EOPatch
        # Besides bands (BANDS) and cloud mask (CLM), the respective bounding box and image timestamps are added.
        # Meta-information about the band name and its resulting order number are also attached.
        eopatch.data['BANDS'] = np.transpose(np.stack(band_sets.to_numpy()), (0,2,3,1))
        eopatch.bbox = BBox(kwargs.get('patch_bbox'), crs=CRS.UTM_33N)
        eopatch.timestamp = self.get_images_dates()
        eopatch.meta_info = {band:order for band, order in zip(CustomInput.BAND_NAMES, CustomInput.BAND_ORDER)}

        return eopatch

    def load_bands(self, row):
        '''
        Loads 10m and 20m Sentinel-2 bands without unzipping the file.
        '''
        zip_name = row[0]+'.zip'

        # Load S2A zipfile as a GDAL Dataset
        S2L2A_image = gdal.Open(os.path.join(self.path_to_images, zip_name))

        # Retrieving the S2A's subdatasets (band groups)
        subdatasets = S2L2A_image.GetSubDatasets()
        l2a10m, l2a20m = subdatasets[0][0], subdatasets[1][0]
        del S2L2A_image

        return l2a10m, l2a20m

    def process_bands(self, l2a10m, l2a20m, clip_patch):
        '''
        Processes 10m and 20m bands using BandOperations class-interface (further down below) and rasterio MemoryFiles.
        '''
        with rasterio.open(l2a10m) as src:
            with BandOperations.clip_by_patch(src, clip_patch) as clipped:
                ten = clipped.read()
        with rasterio.open(l2a20m) as src:
            with BandOperations.upscale(src) as resampled:
                with BandOperations.clip_by_patch(resampled, clip_patch) as clipped:
                    twenty = clipped.read()

        return np.concatenate((ten, twenty), axis=0)[CustomInput.BAND_ORDER,...]

    def get_images_dates(self):
        '''
        Auxiliary function to get image timestamps from their metadata. Ingestion data is used for the timestamp.
        '''
        return self.images.ingestiondate.to_list()
```

```
class AddMask(EOTask):
    '''
    A class to retrieve, process and assign a SCL-based mask to all imagery within EOPatches.
    '''

    def __init__(self, images_gdf, path_to_images, mask_name='CLM'):
        self.images = images_gdf
        self.path_to_images = path_to_images
        self.mask_name = mask_name

    def execute(self, eopatch, **kwargs):
        clms = self.images.apply(lambda row: self.process_clm(self.load_clm(row), kwargs.get('patch_bbox')), axis=1)
        eopatch.mask[self.mask_name] = np.vstack(clms.to_numpy())[..., np.newaxis]

        return eopatch

    def load_clm(self, row):
        '''
        Loads SCL using GDAL vsizip Virtual File.

        This could be similar to loading bands, however, GDAL v2.4.3 Sentinel-2 drivers do not have the capability to load SCL as a subdataset.
This is resolved in GDAL v3.1.0, which could not be easily installed to the server due to dependency issues.
        '''
        zip_name = row[0]+'.zip'
        zz = zipfile.ZipFile(os.path.join(self.path_to_images, zip_name))
        scl_path = [f.filename for f in zz.filelist if f.filename.find('SCL_20m') >= 0][0]

        assert len(scl_path) != 1, 'Unexpected behaviour of the Sentinel-2 image zipfile when trying to load SCL.'
        scl = os.path.join('/vsizip', self.path_to_images, zip_name, scl_path)
        del zz

        return scl

    def process_clm(self, scl, clip_patch):
        '''
        Processes SCL in a similar fashion as process_bands method processes bands.
        '''
        with rasterio.open(scl) as src:
            with BandOperations.upscale(src) as resampled:
                with BandOperations.clip_by_patch(resampled, clip_patch) as clipped:
                    scl = clipped.read()

        return BandOperations.scl_to_mask(scl)


class BandOperations:
    '''
    Auxiliary class-interface for operations with Sentinel-2 bands and SCL layer.
    '''

    # Predetermined reclassification structure for the SCL layer (Baetens et al. 2019)
    SCL_RECLASS = {
        0: False,
        1: True,
        2: True,
        3: False,
        4: True,
        5: True,
        6: True,
        7: True,
        8: False,
```

```
        9: False,
        10: False,
        11: True
    }

    @contextmanager
    def upscale(raster, upscale_factor=2):
        '''
        Upscales Sentinel-2 20m bands to 10 m pixel size.
        '''
        t = raster.transform
        # Rescale the metadata
        transform = Affine(t.a / upscale_factor, t.b, t.c, t.d, t.e / upscale_factor, t.f)
        height = raster.height * upscale_factor
        width = raster.width * upscale_factor
        profile = raster.profile
        profile.update(transform=transform, driver='GTiff', height=height, width=width)

        # Resample data to target shape
        data = raster.read(
            out_shape=(
                raster.count,
                int(raster.height * upscale_factor),
                int(raster.width * upscale_factor)
            ),
            resampling=Resampling.nearest
        )

        with MemoryFile() as memfile:
            with memfile.open(**profile) as dataset:
                dataset.write(data)
                del data
            with memfile.open() as dataset:
                yield dataset

    @contextmanager
    def clip_by_patch(raster, clip_patch):
        '''
        Clips a set of bands by the respective EOPatch's bounding box
        '''
        out_img, out_transform = mask(raster, shapes=[clip_patch], crop=True)
        profile = raster.profile
        profile.update(transform=out_transform, driver='GTiff', height=out_img.shape[1], width=out_img.shape[2])

        with MemoryFile() as memfile:
            with memfile.open(**profile) as dataset:
                dataset.write(out_img)
                del out_img
            with memfile.open() as dataset:
                yield dataset

    def scl_to_mask(scl):
        '''
        Reclassifies SCL to a cloud mask.
        No-data pixels, thus those places where Sentinel-2 scene is not captured, are added to the mask.
        '''
        return np.vectorize(BandOperations.SCL_RECLASS.get)(scl)


class DerivateProduct(EOTask):
    """
```

```python
    Custom EOTask that calculates a user-chosen derivate product (index or similar).
    The band placholders must be called the same name as found in EOPatch.meta_info.
    """

    # Regex expressions to check whether the user-specified eqution contains allowed features (bands)
    ALLOWED_BANDS = re.compile(r"[B][11|12]{2}|[B][8][A]|[B][2|3|4|5|6|7|8]{1}")
    FALSE_BANDS = re.compile(r"[B](?<!\d)[9](?!\d)|[B](?<!\d)[1](?!\d)|[B]10")

    def __init__(self, derivate_name, equation):
        self.derivate_name = derivate_name
        self.equation = equation
        self.extracted_features = set(re.findall(DerivateProduct.ALLOWED_BANDS, self.equation))

    def check_equation(self):
        ''
        Checks if the equation contains correct features.
        ''
        disallowed = set(re.findall(DerivateProduct.FALSE_BANDS, self.equation))
        assert len(self.extracted_features) > 0, 'There are invalid or no band features in the equation'
        assert len(disallowed) == 0, 'Bands 1, 9, 10 are not applicable at this point'

    def equation_features_as_variables(self, band_arrays):
        ''
        Extracts feature names to create the variable-like strings that the Python eval method recognizes
        and processes.
        ''
        features_as_variables = list(map(lambda band: 'band_arrays[\'"+ band +"\']', band_arrays.keys()))
        return dict(zip(band_arrays.keys(), features_as_variables))

    def repopulate_equation(self, feature_variables):
        ''
        Creates a new equation for the Python eval function where user-specified band names are exchanged
        for variable-like names.
        ''
        new_equation = self.equation
        for band, variable in feature_variables.items():
            new_equation = new_equation.replace(band, variable)
        return new_equation

    def execute(self, eopatch):
        # Checking equation
        self.check_equation()

        # Retrieving the right bands with the help of meta info FeatureType mapping, using extracted bands from the equation
        band_arrays = {band:eopatch.data['BANDS'][..., eopatch.meta_info[band]] for band in self.extracted_features}

        # Synthetizing user-specified equation band names with variables to correctly index the band_arrays array
        feature_variables = self.equation_features_as_variables(band_arrays)
        new_equation = self.repopulate_equation(feature_variables)

        # Evaluating the new equation where band names are variable-like strings to retrieve genuine band arrays
        derivate_product = eval(new_equation)

        # Saving multi-image feature to EOPatch
        eopatch.data[self.derivate_name] = derivate_product[..., np.newaxis]

        return eopatch


class MaskValidation:
```

```python
    """
    Vaidation of each SCL mask. If the there are more than a threshold of False pixels, it is removed. Adapted from the method of Lubej (2019a).
    """
    def __init__(self, threshold):
        self.threshold = threshold

    def __call__(self, mask):
        '''
        Return if non-valid pixels constitute less than threshold.
        '''
        valid = np.count_nonzero(mask) / np.size(mask)
        return (1 - valid) < self.threshold


class NanRemover(EOTask):
    '''
    An auxiliary class to remove Nan values from sampled data.
    '''
    def __init__(self, sampled_features_name, sampled_sampled_lulc_name):
        self.sampled_features_name = sampled_features_name
        self.sampled_sampled_lulc_name = sampled_lulc_name

    def execute(self, eopatch):
        '''
        Removes no-data values from sampled data.
        '''
        features = eopatch.data[self.sampled_features_name]
        lulc = eopatch.mask_timeless[self.sampled_lulc_name]
        unique_nans = self.get_all_unique_nan_indices(features)

        new_features = []
        for timeframe in features:
            timeframe_killed_nans = [np.delete(timeframe[...,0,i], unique_nans) for i in range(11)]
            new_features.append(np.stack(timeframe_killed_nans))
        eopatch.data[self.sampled_features_name] = np.transpose(np.stack(np.array(new_features))[np.newaxis], (1,3,0,2))

        new_classes = np.delete(lulc[..., 0, 0], unique_nans)
        eopatch.mask_timeless[self.sampled_lulc_name] = new_classes[..., np.newaxis, np.newaxis]

        return eopatch

    def get_all_unique_nan_indices(self, features):
        '''
        Gets all unique nan indices within NumPy array of sampled features.
        '''
        nan_indices = []
        for timeframe in features:
            indices = [np.argwhere(np.isnan(timeframe[..., 0, band])) for band in range(11)]
            indices = np.unique(np.concatenate(indices).ravel())
            nan_indices.append(indices)
        return np.unique(np.concatenate(nan_indices).ravel())


class EstimatorParser:
    '''
    A class that reduces the last dimensions of the merged-feature time frames and class labels to prepare them for the Scikit-learn estimator.
    '''

    def __init__(self, eopatches, patch_ids, features='FEATURES_SAMPLED', classes='LULC'):
        self.eopatches = eopatches
```

```
        self.patch_ids = patch_ids
        self.features = features
        self.classes = classes

    def merge_features(self):
        '''
        Stacks sampled features from EOPatches on the top each other.
        '''
        f_list = [self.eopatches[pid].data[self.features] for pid in self.patch_ids]
        merged = []
        for i in range(len(f_list)):
            t, px, w, b = f_list[i].shape
            merged.append(f_list[i].reshape(px, t*b))
        return np.concatenate(merged)

    def merge_classes(self):
        '''
        Stacks LULC labels from EOPatches on the top each other.
        '''
        return np.concatenate([self.eopatches[pid].mask_timeless[self.classes][..., 0, 0] for pid in self.patch_ids])

    def __call__(self):
        # Creating a single vector of pixels and labels in the correct order
        merged_features = self.merge_features()
        merged_classes = self.merge_classes()

        return merged_features, merged_classes
```

Figure S5. The pipeline.py source code

```
[ ]: # Custom modules
from aoi import Project, AOI, Patcher
from eolearn_datafill import S2L2AImages, CustomInput, AddMask, DerivateProduct,␣
 ↪MaskValidation, EstimatorParser

# Scikit-learn + LightGBM
import lightgbm as lgb
from sklearn import metrics

# eo-learn + sentinelhub
from eolearn.core import EOTask, EOPatch, EOExecutor, LinearWorkflow,␣
 ↪FeatureType, OverwritePermission, SaveTask, MergeFeatureTask
from eolearn.features import LinearInterpolation, SimpleFilterTask, PredictPatch
from eolearn.geometry import VectorToRaster, PointSamplingTask, ErosionTask
from eolearn.io import ExportToTiff
from sentinelhub import CRS
```

```
[ ]: project = Project('jihozapad_brna')

# Choose study area: must be a geojson or a shapefile of
#a single feautre, which is a polygon of AOI.
aoi = AOI('basedata/jmk_aoi.geojson', crs=CRS.UTM_33N)
aoi.convert_desired_CRS()
print(aoi)

# Splitting the area of interest.
patcher = Patcher(project)
patcher.get_xy_splitters(aoi.dimensions, patch_factor=1)
patcher.split_bboxes(aoi.shape)

# Create a GeoDataFrame of patches made according to the splitters and central␣
 ↪patch selection.
# Initial patch can be selected by subset_patch.
patcher.get_patch_gdf(subset_patch=54, save=False)
patcher.get_patch_map(aoi.gdf)
```

```
[ ]: # Pre-processing cadastre reference map

# Loading this file takes around 3 minutes
LC = gpd.read_file('cadastre_south_moravian_region.shp')

# Copying dataset for modifications
land_cover = LC.copy()

# Reclassifying the information classes
reclassify = {
```

```
    2: 1,
    3: 0,
    4: 2,
    5: 3,
    6: 4,
    7: 5,
    10: 6,
    11: 7,
    13: 8,
    14: 0,
}
land_cover['druh_pozem'] = land_cover['druh_pozem'].map(reclassify) # Note:␣
→druh_pozem = lulc type on the estate


# Masking invalid geometries from the cadastre dataset
mask = (land_cover['geometry']!='shapely.geometry.polygon.Polygon')
land_cover = land_cover[mask]

# Adding roads to built-up areas
land_cover.loc[land_cover.zpusob_vyu.isin([15,16,18]), 'druh_pozem'] = 8

# Removing class 0: no-data/other surfaces
land_cover = land_cover[land_cover['druh_pozem']!=0]
```

[ ]:
```
# Obtaining available Sentinel-2 L2A images and retrieving their metadata
images = S2L2AImages(patcher.gdf, project.FOLDER)
images.get_patches_bbox()
images.get_available(esa_sci_hub_credentials = ('username', 'password'), #␣
→Credentials to Copernicus Open Access Hub must be supplied
                     date_range=('20190301', '20191130'), # Download range
                     cloudcov=(0, 60)) # Cloud coverage restriction

# The possibility to select concrete images from the range
images.select() # No arguments means download all available

# Download images
images.download()
```

[ ]:
```
# NOT IN THE EXAMPLE USAGE EXPERIMENT
# Custom EOTask DerivateProduct for computing derivate products
# Now it statically takes BANDS
ndvi = DerivateProduct('NDVI', # Derivate product name
                       '(B4-B8)/(B4+B8)' # Computing formula
                      )
```

[ ]:

```
# Desining the workflow to amend and process Sentinel-2, cadastre data
# and to almost prepare them for the classification

# Custom EOTask for adding and parsing downloaded images.
customS2L2A = CustomInput(images_gdf=images.downloaded, # Downloaded images␣
↪dataframe
                          path_to_images=project.FOLDER, # Path to downloaded␣
↪images
                          custom_input_name='BANDS') # Key name of input data in␣
↪EOPatches


# Custom EOTask for adding a mask from Sentinel-2 L2A 20m Scene Classification␣
↪Layer (SCL)
# that is clipped and resampled to 10 m
addmask = AddMask(images_gdf=images.downloaded,
                  path_to_images=project.FOLDER,
                  mask_name='SCL_MASK')

# Original eo-learn EOTask for rasterizing cadastre LULC map
# and adding it to FeatureType.mask_timeless['LULC']
lulc_rasterization = VectorToRaster(
    land_cover, # Land cover GeoDataFrame
    (FeatureType.MASK_TIMELESS, 'LULC'), # FeatureType and name of the feature␣
↪to save to EOPatches
    values_column='druh_pozem', # GeoDataFrame column to be used as a raster␣
↪value
    raster_shape=(FeatureType.MASK, 'SCL_MASK'), # Make land cover to have same␣
↪dimensions and cell size as SCL MASK
    raster_dtype=np.dtype(np.uint8) # NumPy array data type = unsigned 8 bit␣
↪integer (meaningful for cadastre data)
    )

# Original eo-learn EOTask for merging data along the band dimension
merging = MergeFeatureTask({FeatureType.DATA: ['BANDS', 'NDVI']}, # Features to␣
↪merge
                          (FeatureType.DATA, 'FEATURES') # FeatureType and name␣
↪of the feature to save to EOPatches
                         )

# Validation, according to the SCL mask
valid_data = MaskValidation(0.1) # Maximum threshold (10 %) of False pixels

# Original eo-learn for filtering images according to the MaskValidation
filtering = SimpleFilterTask((FeatureType.MASK, 'SCL_MASK'), # FeatureType and␣
↪name of the feature to save to EOPatches
```

```
                             valid_data # CLMValidation results
                            )

# Original eo-learn for merging data along the band dimension
interpolation = LinearInterpolation(
    'FEATURES', # FeatureType data to be interpolated
    copy_features=[
        (FeatureType.MASK_TIMELESS, 'LULC'),
        (FeatureType.META_INFO)
    ], # Preserving some features in EOPatches
    mask_feature=(FeatureType.MASK, 'SCL_MASK'), # Masking data with the␣
 ↪respective CLMs
    resample_range=('2019-03-30', # First date of arbitrary range
                    '2019-10-18', # Last date of arbitrary range
                     10 # Interpolation step in days
                   )
    )

# Sampling pixels from patches for the estimator
spatial_sampling = PointSamplingTask(
    n_samples=50000, # Number of pixels to sample from each band in each time␣
 ↪frame in each EOPatch
    ref_mask_feature='LULC', # Reference map (e.g. cadastre reference map)
    ref_labels=list(range(1,9)), # Unique information class labels from the␣
 ↪reference map
    sample_features=[  # Specify which fields to sample
        (FeatureType.DATA, 'FEATURES'),
        (FeatureType.MASK_TIMELESS, 'LULC')
    ])

# Custom EOTask for removing no-date (Nans) from sampled pixels
nrm = NanRemover(sampled_feature_name='FEATURES',
                 sampled_lulc_name='LULC_SAMPLED'
                )

# Original eo-learn EOTask for saving EOPatches as npy files to disk
save = SaveTask(project.FOLDER, overwrite_permission=OverwritePermission.
 ↪OVERWRITE_PATCH)
```

[ ]:
```
# Instantiating EOWorkflow
workflow = LinearWorkflow(
    customS2L2A,
    addmask,
    ndvi,
    ndwi,
    ndbi,
    lulc_rasterization,
```

```
    merging,
    filtering,
    interpolation,
    lulc_erosion,
    spatial_sampling,
    nrm,
    save
)
```

[ ]:
```
# External parameters of the workflow
execution_args = patcher.gdf.apply(lambda row: {
        customS2L2A: {'patch_bbox': row[3]}, # row[3] = Patcher DataFrame␣
 ↪position of bounding box column
        addmask: {'patch_bbox': row[3]},
        save: {'eopatch_folder': f'eopatch{row[2]}'} # row[2] = Patcher␣
 ↪DataFrame position of bounding box ID column
    }, axis=1).to_list()

# Instantiating EOExecutor
executor = EOExecutor(workflow, # The workflow defined above
                      execution_args, # External execution arguments to feed to␣
 ↪EOTasks
                      save_logs=True, # Save detailed logs about what is␣
 ↪happening
                      logs_folder=project.FOLDER)

# Running the workflow
executor.run(workers=9) # Specify multiprocessing division to CPU threads
executor.make_report() # Make an HTML report
```

[ ]:
```
# Loading EOPatches and feeding them to EstimatorParser class to get
# the feature vectors and information class labels
eopatches = np.array([EOPatch.load(os.path.join(project.FOLDER, f'eopatch{i}'),␣
 ↪lazy_loading=True) for i in range(9)])

# Reducing feature space of training data
parse_training_data = EstimatorParser(eopatches, # Feed EOPatches
                      patch_ids=[7,3,5,8,0], # Which EOPatches
                      features='FEATURES_SAMPLED', # Which features to reduce
                      classes='LULC_SAMPLED') # LULC to reduce

# Reducing feature space of testing data
parse_testing_data = EstimatorParser(eopatches,
                      patch_ids=[6,1,2,4],
                      features='FEATURES_SAMPLED',
                      classes='LULC_SAMPLED')
```

```
train_features, train_classes = parse_training_data()
test_features, test_classes = parse_testing_data()
```

[ ]:

```
# Adapted from Lubej (2019a, 2019b), eo-learn (2018)
#Set up training classes
labels_unique = np.unique(train_classes)

# LightGBM model with default parametres
model = lgb.LGBMClassifier(
    objective='multiclassova',
    num_class=len(labels_unique),
    metric='multi_logloss',
    randoma_state = 10
)

# Train the model
model.fit(train_features, train_classes)

# Testing the trained model on test features
prediction_test_set = model.predict(features_test)

# Obtaining pixel confusion matrix from predicted test samples and
# corresponding (but ground truth) test classes
confusion_matrix = metrics.confusion_matrix(prediction_test_set, test_classes)
```

[ ]:

```
# Showcase of predicting an EOPatch
predict = PredictPatch(model, feature_name='PREDICTED_LULC')
eopatch_predicted = PredictPatch.execute(eopatch)

# Original EOTask for exporting predicted LULC to GEoTiff
export_tiff = ExportToTiff((FeatureType.MASK_TIMELESS, 'PREDICTED_LULC'),␣
 ↪filename='predicted_eopatch1.tiff')
export_tiff.execute(eopatch_predicted)
```

Figure S6. Jupyter Notebook with the implementation applied to the example usage experiment