**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.

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

```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
        derivatate_product = self.arbitrary_method(self.equation)
        # Load or alter data to EOPatch
        eopatch.data[self.derivate_name] = derivatate_product

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

Figure S3. A sample EOWorkflow and EOExecutor as used to pipeline EOTasks. 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
    derivatate_product,  # Instance of DerivateProduct EOTask
    save_eopatches
)

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

EOWorkflows (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 EOWorkflows, 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.
import sys
import os
import datetime
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gp
d
from shapely.geometry import Polygon, box
d
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 [Y] or create a new one [N]?")

        if input(decision) == 1:
            print(f"Project (path) has been set to be used.")
            return Project.set_existing_project(path)
        decision = 2:
            print(f"Change the project name at project instance __init__")
            return None
        path = input(f"Project (path) created at: (path)")
        return path

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

@staticmethod
def set_existing_project(path):
    """
    Gets the existing project path if it exists.
    """
    for folder in os.listdir(path):
        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.aoi = self.set_AOI_aoi()
        self.shape = self.set_AOI_shape()
        self.dimensions = self.set_AOI_dimensions()

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

    def set_AOI_aoi(self):
        # Loads a GEOJSON or Shapefile to a GEOJSON or Shapefile.
        return gpd.read_file(self.path)

    def set_AOI_crs(self):
        return crs

    def set_AOI_shape(self):
        # Extracts shape from the GEOJSON.
        return self.aoi.geometry.values[0]

    def set_AOI_dimensions(self):
        # Obtains dimensions of the AOI.
        shape = self.shape

    def convert_desired_CRS(self):
        # For safety, the AOI it 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.aoi = self.aoi.to_crs({'init': 'epsg:4326', 'crs': self.crs})

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

class Patcher:
    # A class that splits the AOI into patches based on the given dimensions. Creates list of layers and info list with parent key (aoi of the patch) and spatial indexes of patches.
def init(self, project_folder, crs=CRS_UTM_33N):
    self.crs = crs
    self.project_folder = project_folder
    self.subset = None
    self.bbox_list = None
    self.info_list = None
    self.patch_id = None
    self.patch_bbox = None

    def get_xy_splits(self, bnd_box, patch_factor):
        x, y = bnd_box
        while x >= 100 or y >= 100:
            x = x // 2
            y = y // 2
        return x, y
    return x, y

    def split_bbox(self, shape):
        bbox_spliter = BBoxSPLInter(shape, self.crs, self.xy_splits)
        self.bbox_list = bbox_spliter.get_bbox_list()
        self.info_list = bbox_spliter.get_info_list()

    def get_patch_ids(self, subset, patch=None, save=False):
        subset = self.select_patch_subset(subset, patch)
        geometry = [bbox.get_polygon() for bbox in self.bbox_list]
        id_list = [info[4] for info in self.info_list]
        df = pd.DataFrame({'bbox': geometry, 'id': id_list})
        crs = CRS.from_string(self.crs)
        geometry = geometry
        df = df.apply(lambda row: Polygon(row.geometry.centroid), axis=1)
        self.id_list = df
        if save:
            self.save_patches_as_shape()

    def select_patch_subset(self, subset, ID):
        if ID is None:
            return None
        aux = [bbox for bbox in self.bbox_list]
        aux = [bbox for bbox in self.bbox_list if bbox[4] == ID]
        if len(aux) == 1:
            aux = aux[0]
        else:
            # Renumbering the patches
Figure S4. The aoi.py module source code
class Sentinel2L2AImages:

    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_id, path_to_download):
        self.patch_id = patch_id
        self.path_to_download = path_to_download
        self.box = None
        self.ordered = pd.DataFrame()
        self.downloaded = pd.DataFrame()
        self.url = None
        self.products = None

    def get_patches_box(self):
        ""
        Gets total bounding box of selected patches for downloading Sentinel2L2A images
        through sentinel2 API. Only adjacent or very close patches should be chosen for now.
        ""
        self.box = box("self.patch_id", "EPSG:4326")
        return self.box

    def get_available(self, esa_sci_hub_credentials = (username, password), data_range = (20190629, 20190627),
                      cloud_percentage = 0):

According to the user-specified parameters, available Sentinel-2 L2A images are obtained from Sentinel2 API.

```python
assert self.bbox != None. Bbox for patches is empty. Use self.set_patches_bbox to obtain one,
self.api = Sentinel2API("geo-sci_hub_credential")
self.products = self.api.get_product(self.bbox,
date_range=..., platform_name="Sentinel-2",
cloudcover_percentage=cloudcover_percentage=cloudcov
products=['S2MSI2A'])
assert not self.products == None. There were no sentinel2 api or available images found.
self.available = self.api_to_geodatabase(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") in selected)
        self.downloaded = 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 sentinel2 api or available images found.
    self.downloaded = self.api.download(self, axes=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["zip"]
    if not os.path.isfile(zip_file):
        self.api.download_image(image_row["path_to_image"])]
        directory_path = self.path_to_download

class CustomInput(EO2Task):
    """A class that prepares Sentinel-2 images to further work in the pipeline.
""
    # Bands in the original Sentinel-2 zipfile (SAFE file) are processed 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 FeatureTable.
    BAND_ORDER = np.array([7, 1, 0, 6, 3, 4, 5, 8, 9])

    def init(self, images, self.path_to_images, custom_input_name):
        self.images = images
```
def execute(self, **kwargs):
    """
    Mandatory EOTask method, orchestrator for the Sentinel-2 imagery processing, using the methods below.
    """
    # Initializing a new EOBatch
    eobatch = EOBatch()

    # Processing S2L2A data and CLM
    band_sets = self.images.getBandSets(timedelta(days=1), self.process.bands(self.load_bands(row), **kwargs), axis=1)

    # Adding new features to EOBatch
    # Besides bands (BANDS) and cloud mask (CLM), the respective bounding box and image timestamps are added.
    eobatch.bands = np.transpose(np.array(band_sets, (0,2,1)), (0,2,1,3))
    eobatch.box = BBox(bands.get('band_box'), CRS=CRS_UTM_31N)
    eobatch.timestamps = self.set_images_data()
    eobatch.meta_info = (bands.order_for_band, order in zip(CustomInput.BAND_NAMES, CustomInput.BAND_ORDER))

    return eobatch

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

    # Load S2A file 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_match):
    """
    Processes 10m and 20m bands using BandOperations class-interface (further down below) and rasterio MemoryFiles.
    """
    with rasterio.open(l2a10m, as_src=True) as src:
        with BandOperations.clip_by_patch(src, clip_match) as clipped:
            ten = clipped.read()
    with rasterio.open(l2a20m, as_src=True) as src:
        with BandOperations.resamp(src, as_resampled, clip_match) as clipped:
            twenty = clipped.read()

    return np.concatenate((ten, twenty), axis=0)

def set_images_data(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(KOTask):
    
    A class to retrieve, process and assign a SCL-based mask to all imagery within KOBand.
    
    def __init__(self, images, path_to_images, mask_name, "scl":
        self.images = images
        self.path_to_images = path_to_images
        self.mask_name = mask_name

    def execute(self, approach, "approach":
        Class = self.imagesapply(lambda row: self.process_clip(self, clip(row)), axis=1)
        approach = np.stack((mask_name) + [np.stack((class_to_mosaic() for _ in range(1)))]
        approach = approach

    return approach

    def load_clip(self, row):
        
        Loads SCL using GDAL using 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[1] + "scl"
        zipf = zipfile.ZipFile(os.path.join(self.path_to_images, zip_name))
        scl_list = [f for f in zipfile if f.endswith("SCL_20m")]
        
        assert len(scl_list) == 1, 'Unexpected behaviour of the Sentinel-2 image file when trying to load SCL.'
        scl = os.path.join(os.path.join(self.path_to_images, zip_name, scl_list[0])
        
    return scl

    def process_clip(self, self, scl, clip:
        
        Processes SCL in a similar fashion as process bands method processes bands.

        with rasterio.open(scl) as src:
            with BandOperations.resample(src) as resampled:
                with BandOperations.clip_by_patch(resampled, clip) as clipped:
                    
    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 (Barros 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):
    """
    Upscales Sentinel-2 20m bands to 10 m pixel size.
    """
    t = raster.transform
    # Recalculate the metadata
    transform = AffineTransform(upscale_factor, 0, 0, upscale_factor, 0, 0)
    height = raster.height * upscale_factor
    width = raster.width * upscale_factor
    profile = raster.profile
    profile.update(transform=transform.spatial, driver="GTiff", height=height, width=width)

    # Resample data to target shape
    data = raster.read()
    out_shp =
        (raster.crs, upscale_factor * raster.height, upscale_factor * raster.width),
        resampling=Resampling.nearest
    }
    with MemoryFile() as memfile:
        with memfile.open(""", profile) as dataset:
            dataset.write(data)
            del data
        with memfile.open(""", profile) as dataset:
            yield dataset

@contextmanager
def clip_by_patch(raster, clip_patch):
    """
    Clips a set of bands by the respective EO patch’s bounding box
    """
    out_img = raster.copy(transform=(patch, shape=[clip_patch, crop=True])
    profile = raster.profile
    profile.update(transform=patch, driver="GTiff", height=patch.height, width=patch.width)

    with MemoryFile() as memfile:
        with memfile.open(""", profile) as dataset:
            dataset.write(out_img)
        with memfile.open(""", profile) as dataset:
            yield dataset

def scl_to_mask(scl):
    """
    Reclassifies SCL to a cloud mask
    No-data pixels, thus those places where Sentinel-2 scans are not captured, are added to the mask
    """
    return ow.tiftools.ReadOperations.SCL_RECLASSASET(scl)

class GenerateProduct(EOTask):
    """
Custom EOTask that calculates a user-chosen derivative product (index or similar). The band placeholders must be called the same name as found in \texttt{EOPatch.meta.info}.

# Regex expressions to check whether the user-specified equation contains allowed features (bands)
ALLOWED_BANDS = re.compile(r"[^B][1][1][1][0][2][D][B][A][D][2][9][4][5][6][7][8][1][1][0][0]"
FALSE_BANDS = re.compile(r"[^B][1][1][1][0][2][D][B][A][D][2][9][4][5][6][7][8][1][1][0][0]


def init(self, derivative_name, equation):
    
    self.derivative_name = derivative_name
    self.equation = equation
    self.extracted_features = set(re.findall(DerivativeProduct.ALLOWED_BANDS, self.equation))


def check_equation(self):
    
    # Checks if the equation contains correct features.
    
    disallowed = set(re.findall(DerivativeProduct.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 pour


def equation_features_as_variables(self, band_array):
    
    # Extracts feature names to create the variable-like strings that the Python eval method recognizes
    
    features_as_variables = list(map(lambda band: band.array[0] + "=" + band.array[1], band.array[2]))
    return dict([band.array[2], features_as_variables])


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


def evaluate(self, epatch):
    
    # Checking equation
    self.check_equation()

    # Retrieving the right bands with the help of meta info EOPatch mapping, using extracted bands from the equation
    band_array = [feature_variables[band] for band in self.extracted_features]

    # Synthesizing user-specified equation band names with variables to correctly index the band array
    feature_variables = self.equation_features_as_variables(band_array)
    new_equation = self.equation_equation(feature_variables)

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

    # Saving multi-image (raster to EOPatch)
    epatch.add_content(derivative_name) = derivative_product[ ... np.newaxis]

    return epatch


class MaskValidation
version of each SCL mask. If the there are more than a threshold of False pixels, it is removed. Adapted from the method of (2011a).

```python
def __init__(self, threshold):
    self.threshold = threshold

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

class NanRemover(EO7mask):
    """An auxiliary class to remove Nan values from sampled data."

def __init__(self, sample_features_name, sample_sampled_huc_name):
    self.sampled_features_name = sample_features_name
    self.sampled_sampled_huc_name = sample_sampled_huc_name

def remove(self, sample):
    """Removes no-data values from sampled data"
    features = sample.data[self.sampled_features_name]
    huc = sample.mask_time_series(self.sampled_huc_name)
    unique_num = self.set_all_unique_nan_indices(features)

    ssw_features = []
    for timeframe in features:
        timeframe_killed_nan = np.delete(timeframe[~huc, :], unique_num) for i in range(11)]
        ssw_features.append(timeframe_killed_nan)
    sample.mask_time_series(self.sampled_huc_name) = np.concatenate(ssw_features)

    new_classes = np.delete(huc, [0.0, 0.0]) unique_three
    sample.mask_time_series(self.sampled_huc_name) = new_classes[~np.isnan(huc)]

    return sample

    def set_all_unique_nan_indices(self, feature):
        """Gets all unique nan indices within NumPy array of sampled features"

    ssw_ind = []
    for timeframe in features:
        indices = np.delete(timeframe, timeframe[~huc, :]), for band in range(11)]
        indices = np.unique(indices)
        indices.append(indices)
        ssw_indices.append(indices)
    return np.unique(np.concatenate(ssw_ind), nan_indices), nan_indices
```

class EstimateDates:
    """A class that reduces the last dimensions of the merged-feature time series and class labels to prepare them for the scikit-learn estimator"

def __init__(self, sample_features, sample_labels, features=FEATURES_SAMPLED, classes=LUCLC):
    self.evaluation = evaluation
```
Figure S5. The pipeline.py source code
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 EDTask, EDPatch, EDExecutor, 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('jihozapat_brna')

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

# Splitting the area of interest.
patcher = Patcher(project)
patcher.get_xySplitters(ai.dimensions, patch_factor=1)
patcher.split_bboxs(ai.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(ai.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 = {

land_cover['drum_base'] = land_cover['drum_base'].map(reclassify) # Note:
# drum_base = lake type on the estate

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

# Adding roads to built-up areas
land_cover['is_road'] = land_cover.spatial_year.isin([15, 16, 18])

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

# Obtain available Sentinel-2 L2A images and retrieving their metadata
images = EO2Images(patcher,pool,project,FOLOEX)
images.get_patches_base()
images.get_available(username, password),

# A valid user login to Copernicus Open Access Hub must be supplied

date_range=('20190101', '20191130'), # Download range
cloudcov=(0, 60) # Cloud coverage restriction

# The possibility to select concrete images from the range
images.select() # So arguments mean download all available

# Download images
download_images(images)

# NOT IN THE EXAMPLE USAGE EXPERIMENT
# Custom EO2Task DerivateProduct for computing derivative products
# Note: it statically takes band of
mwi = DerivateProduct('MVI', # Derivate product name
'(B4-B8)/(B4+B8)' # Computing formula
)
## Describing the workflow to amend and process Sentinel-2, cadastral data

### Custom ETTask for adding and parsing downloaded images

```python
customD2Li2a = CustomInput(image_gdf=image downloaded, # Downloaded images,
                           dataframe=
                           path_to_images=project, FOLDER, # Path to downloaded,
                           images=
                           custom_input_name='LAND') # Key name of input data in,
                           # EOPatches
```

### Custom ETTask for adding a mask from Sentinel-2 Li2 20m Scene Classification

```python
# That is clipped and resampled to 10 m
addmask = AddMask(images_gdf=image downloaded,
                   path_to_images=project, FOLDER, mask_name='SCL_MASK')
```

### Original re-learn ETTask for rasterizing cadastral ULC map

```python
# and adding it to FeatureType.mask_timeless('ULC')
lulc_rasterization = VectorToRaster(
    land_cover, # Land cover GeoDataFrame
    FeatureType(MARK, TIMELESS, 'ULC'), # FeatureType and name of the feature,
    to save to EOPatches,
    values_column='dran_pozem', # GeoDataFrame column to be used as a raster;
    value=raster_shape=(FeatureType.MARK, 'SCL_MASK'), # Make land cover to have same
    dimensions and cell size as SCL MASK
    raster_dtype=np.dtype('float64') # NumPy array data type - unsigned 8 bit,
    integer (meaningful for cadastral data)
)
```

### Original re-learn ETTask for merging data along the band dimension

```python
# Original re-learning ETTask for merging data along the band dimension
merging = MergeFeatureTask(FeatureType.DATA=[BANKS, EVENT], # Features to,
                           #save to EOPatches,
                           (FeatureType.DATA, 'FEATURES') # FeatureType and name,
                           # of the feature to save to EOPatches
)
```

### Validation, according to the SCL mask

```python
valid_data = MaskValidation(0.1) # Maximum threshold (10 %) of False pixels
```

### Original re-learn for filtering images according to the MaskValidation filter

```python
filtering = SimpleFilterTask(FeatureType.MARK, 'SCL_MASK', # FeatureType and,
                             None of the feature to save to EOPatches
)
valid_data # CLMValidation results

# Original co-learns 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 EDpackets
    mask_feature=(FeatureType.MASK, 'BCL_MASK'), # Masking data with this
    --respective CLS
    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=10000, # Number of pixels to sample from each band in each time
    frame_in_each_EDPatch
    ref_mask_feature='LULC', # Reference map (e.g. cadastral 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 EDTask for removing no-data (Nans) from sampled pixels
arm = NanRemover(sampled_feature_name='FEATURES',
    sampled_lulc_name='LULC_SAMPLED'
)

# Original co-learn EDTask for saving EDpatches as npy files to disk
save = SaveTask(project.FOLDER, overwrite_permission=OverwritePermission.
    --OVERWRITE_PATCH)

[ ]: # Instantiating EDWorkflow
workflow = LinearWorkflow(
customSZL2A,
    addmask,
    ndvi,
    ndwi,
    ndni,
    lulc_rasterization,
merging,
filtering,
interpolation,
lulc_erosion,
spatial_sampling,
nrm,
save
)

# External parameters of the workflow execution_args = patcher.gdf.apply(lambda row: {
    custom2D2A: {'bbox': row[3]}, # row[3] = Patcher DataFrame
    addmask: {'bbox': row[3]},
    save: {'epatch_folder': f'epatch_{row[2]}'} # row[2] = Patcher
}, axis=1).to_list()

# Instantiating EDExecutor executor = EDExecutor(workflow, # The workflow defined above execution_args, # External execution arguments to feed to
--EDTasks
save_logs=True. # Save detailed logs about what is
--happening
logs_folder=project 'FULL.EXE'

# Running the workflow executor.run(workers=0) # Specify multiprocessing division to CPU threads executor.make_report() # Make an HTML report

# Loading EDPatches and feeding them to EstimatorParser class to get # the feature vectors and information class labels epatches = np.array([EDPatch.load(os.path.join(project 'FULL.EXE', f'epatch{i}')) for i in range(5)])

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

# Reducing feature space of testing data parse_testing_data = EstimatorParser(epatches,
patch_ids=[6,1,2,4],
features='FEATURES_SAMPLED',
classes='LULC_SAMPLED')
Figure S6. Jupyter Notebook with the implementation applied to the example usage experiment.