B

Code of Project

All code implemented is publicly accessible through a Git-lab repository located at following link: http

Listing B.1: Code for NCSS: Creation of table "ids_loc" through the "sqlite3" command using the machine's terminal. Second command counts the number of rows

```sql
CREATE TABLE ids_loc AS
SELECT *
FROM lab_layer
JOIN lab_site ON lab_layer.site_key = lab_site.site_key;

--Counting ids_loc rows
SELECT COUNT(*) FROM ids_loc;
```

Listing B.2: Code for NCSS: Creation of table "ids_loc_prop" through the "sqlite3" command using the machine's terminal. Second command counts the number of rows.

```sql
--Appending lab_chemical_properties table
CREATE TABLE ids_loc_chem AS
SELECT *
FROM ids_loc
JOIN lab_chemical_properties
ON ids_loc.labsampnum = lab_chemical_properties.labsampnum
AND ids_loc.layer_key = lab_chemical_properties.layer_key;

--Appending lab_physical_properties table
CREATE TABLE ids_loc_chem_phy AS
SELECT *
FROM ids_loc_chem
JOIN lab_physical_properties
ON ids_loc_chem.labsampnum = lab_physical_properties.labsampnum
AND ids_loc_chem.layer_key = lab_physical_properties.layer_key;

--Appending lab_major_and_trace_elements_and_oxides table
CREATE TABLE ids_loc_prop AS
SELECT *
FROM ids_loc_chem_phy
JOIN lab_major_and_trace_elements_and_oxides
ON ids_loc_chem_phy.labsampnum = lab_major_and_trace_elements_and_oxides.labsampnum
AND ids_loc_chem_phy.layer_key = lab_major_and_trace_elements_and_oxides.layer_key;

--Counting ids_loc_prop rows
SELECT COUNT(*) FROM ids_loc_prop;
```

Listing B.3: Code for NCSS: Creation of table ”ids_loc_prop_time” through the ”sqlite3” command using the machine’s terminal. Second command counts the number of rows

```sql
CREATE TABLE ids_loc_prop_time AS
SELECT *
FROM ids_loc_prop
JOIN lab_pedon ON ids_loc_prop.pedon_key = lab_pedon.pedon_key
AND ids_loc_prop.site_key = lab_pedon.site_key;

--Counting ids_loc rows
SELECT COUNT(*) FROM ids_loc_prop_time;
```

Listing B.4: Code for NCSS: Creation of table ”data” through the ”sqlite3” command using the machine’s terminal.

```
CREATE TABLE data AS
SELECT labsampnum, pedlabsampnum, hzn_top, hzn_bot, observation_date,
latitude_std_decimal_degrees, longitude_std_decimal_degrees, total_carbon_ncs,
total_nitrogen_ncs, phosphorus_bray1, phosphorus_bray2, phosphorus_major_element,
phosphorus_trace_element, k_nh4_ph_7, potassium_major_element, ca_nh4_ph_7,
calcium_major_element, mg_nh4_ph_7, magnesium_major_element, total_sulfur_ncs,
copper_trace_element, fe_ammoniumoxalate_extractable, iron_sodium_pyro_phosphate,
iron_major_element, manganese_ammonium_oxalate, manganese_dithionite_citrate,
manganese_kcl_extractable, manganese_major_element, manganese_trace_element,
molybdenum_trace_element, zin_trace_element
FROM ids_loc_prop_time;
```

Listing B.5: Code for NCSS: dropping rows with missing latitude, longitude or date

```
import pandas as pd
# DataFrame with NCSS data
df = pd.read_csv('./NCSS/data.csv')
df = df.dropna(subset=['observation_date', 'latitude_std_decimal_degrees', '
    longitude_std_decimal_degrees'])
```

Listing B.6: Code for NCSS: depth parsing

```
df = df.dropna(how='all', subset=['hzn_top', 'hzn_bot'])
# Filter rows where more than 50% of the depth interval is below 30
df = df[(30 - df['hzn_top']) / (df['hzn_bot'] - df['hzn_top']) >= 0.5]
# Drop duplicates based on 'pedlabsampnum' keeping the one with the lowest '
    hzn_top'
df = df.sort_values(by='hzn_top').drop_duplicates(subset='pedlabsampnum',
    keep='first')
```

Listing B.7: Code for NCSS: nutrient conversion to ppm and averaging

```
# Features that need gravimetric to ppm convertion
gravimetric_features = (
    'total_carbon_ncs',
    'total_nitrogen_ncs',
    'total_sulfur_ncs',
    'fe_ammoniumoxalate_extractable',
    'iron_sodium_pyro_phosphate',
    'manganese_dithionite_citrate'
    )

# Features that need meq/100g to ppm convertion
meq_per_100_features = {
    'k_nh4_ph_7': 39.0,
    'ca_nh4_ph_7': 40.08,
    'mg_nh4_ph_7': 24.305
}

# Applying convertions according to README.md file
# Convert all features in gravimetric percentage
for feature in gravimetric_features:
    df[feature] = df[feature].apply(lambda x: round(x * 10**4, 3))
# Convert all features in meq/100g
for feature in meq_per_100_features:
    df[feature] = df[feature].apply(lambda x: round(x * 10 * meq_per_100
        _features[feature], 3))

# Compute average between features that tell the same info. nan values are
    removed from equation
df['P'] = df[['phosphorus_bray1', 'phosphorus_bray2', '
    phosphorus_major_element', 'phosphorus_trace_element']].mean(axis=1)
df['K'] = df[['k_nh4_ph_7', 'potassium_major_element']].mean(axis=1).round(3)
df['Ca'] = df[['ca_nh4_ph_7', 'calcium_major_element']].mean(axis=1).round(3)
df['Mg'] = df[['mg_nh4_ph_7', 'magnesium_major_element']].mean(axis=1).round(
    3)
df['Fe'] = df[['fe_ammoniumoxalate_extractable', 'iron_sodium_pyro_phosphate'
    , 'iron_major_element']].mean(axis=1).round(3)
df['Mn'] = df[['manganese_ammonium_oxalate', 'manganese_dithionite_citrate',
    'manganese_kcl_extractable', 'manganese_major_element', '
    manganese_trace_element']].mean(axis=1).round(3)
df['Cec'] = df[['cec7_clay_ratio', 'cec_nh4_ph_7']].mean(axis=1).round(3)

```

```python
# Renaming
df = df.rename(columns={
    'observation_date': 'date',
    'latitude_std_decimal_degrees': 'latitude',
    'longitude_std_decimal_degrees': 'longitude',
    'total_carbon_ncs': 'C',
    'total_nitrogen_ncs': 'N',
    'total_sulfur_ncs': 'S',
    'copper_trace_element': 'Cu',
    'molybdenum_trace_element': 'Mo',
    'zinc_trace_element': 'Zn'
})

# Keeping necessary features
df = df[[
    'labsampnum', 'pedlabsampnum', 'date',
    'latitude', 'longitude',
    'C', 'N', 'P', 'K', 'Ca', 'Mg', 'S', 'Cu', 'Fe', 'Mn', 'Mo', 'Zn','Cec'
    ]]
```

Listing B.8: Code for NCSS: nutrient missing values and percentage

```python
nutrients = ['C', 'N', 'P', 'K', 'Ca', 'Mg', 'S', 'Cu', 'Fe', 'Mn', 'Mo', 'Zn
    ']
# Calculate missing value count per feature
missing_count_per_feature = df[nutrients].isnull().sum()
# Calculate missing value percentage per feature
missing_percentage_per_feature = round((missing_count_per_feature / len(df))
    * 100, 2)
# Create a DataFrame to display the results
pd.DataFrame({
    'Missing Count': missing_count_per_feature,
    'Missing Percentage': missing_percentage_per_feature
}).transpose()
```

Listing B.9: Code for NCSS: removing points with no nutrient data

```python
df = df.dropna(how='all', subset=nutrients)
```

Listing B.10: Code for NCSS: count data for each satellite's life span

```python
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d').dt.date

mask1 = (df['date'] >= datetime.date(1972, 7, 1)) & (df['date'] <= datetime.
    date(1992, 10, 31))
mask2 = (df['date'] >= datetime.date(2012, 6, 1)) & (df['date'] <= datetime.
    date(2013, 1, 31))

time_interval = [
    mask1 | mask2,
    (df['date'] >= datetime.date(1982, 7, 1)) & (df['date'] <= datetime.date(
        2012, 5, 31)),
    (df['date'] >= datetime.date(1999, 4, 1)),
    (df['date'] >= datetime.date(2013, 2, 1)),
    (df['date'] >= datetime.date(2017, 1, 1))
    ]

MSS = df[time_interval[0]]
TM = df[time_interval[1]]
ETM = df[time_interval[2]]
OLI = df[time_interval[3]]
S2 = df[time_interval[4]]
```

Listing B.11: Code for NCSS: EOS extraction from LMSS, LTM and LETM

```python
import terrasensetk as tstk
import sentinelhub as sh

# Sentinel hub credentials
config = sh.SHConfig()
# Id of the client
config.sh_client_id = "------------------------"
# Sectret token for client
config.sh_client_secret = "----------------------"
config.save()

# Reading shapefile
shp_path = '../NCSS/data.shp'
nutrients = ['C', 'N', 'P', 'K', 'Ca', 'Mg', 'S', 'Cu', 'Fe', 'Mn', 'Mo', 'Zn
    ']
# Landsat 1-5 MSS
mss_down = tstk.LandsatMSSDownloader(config=config, shapefile=shp_path,
    targets=nutrients)
# Landsat 4-5 TM
tm_down = tstk.LandsatTMDownloader(config=config, shapefile=shp_path, targets
    =nutrients)
# Landsat 7 ETM+
etm_down = tstk.LandsatETMDownloader(config=config, shapefile=shp_path,
    targets=nutrients)

downloaders = [mss_down, tm_down, etm_down]
for d in downloaders:
    d.download(path=f'./NCSS/EOPatches/{d.name}',
               date_fmt='%d/%m/%Y', padding_days=10, maxcc=0.1, res=20,
                   report=True)
```

Listing B.12: Code for LUCAS: EOS extraction from S2

```python
import terrasensetk as tstk
import sentinelhub as sh

# Sentinel hub credentials
config = sh.SHConfig()
# Id of the client
config.sh_client_id = "-----------------------"
# Sectret token for client
config.sh_client_secret = "---------------------"
config.save()

# Reading shapefile
shp_path = '../LUCAS/data.shp'
nutrients = ['N', 'P', 'K']
```

```python
# Sentinel S2-A and S2-B
s2_down = tstk.Sentinel2Downloader(config=config, shapefile=shp_path, targets
    =nutrients)
# Downloading
s2_down.download(path=f'./NCSS/EOPatches/{d.name}',
         date_fmt='%d/%m/%Y', padding_days=10, maxcc=0.1, res=20, report=True)
```

Listing B.13: Code for LUCAS: count and percentage number of missing nutrients

```python
import pandas as pd

eopatches_path = '../LUCAS/EOPatches/sentinel-2'
# Creates dataset from EOPatches
dataset = tstk.Dataset(eopatches_path)

df = dataset.get_df()
# Forced convertion to float. < LOD string beomces nan
df.loc[:, labels] = df[labels].apply(lambda x: pd.to_numeric(x, errors='
    coerce'))
df[labels] = df[labels].astype(float)

pd.DataFrame(data=(df[labels].count(), df.shape[0] - df[labels].count()))
```

Listing B.14: Code for LUCAS: scatterplot for N, P and K

```python
import matplotlib.pyplot as plt

for l, name in zip(labels, ('Nitrogen', 'Phosphorus', 'Potassium')):
    values = df[l]
    plt.scatter(range(values.shape[0]), values, label=name)
    plt.title(f'Distribution of {name} ({l})')
    plt.xlabel('Index')
    plt.ylabel(f'{l} (ppm)')
    plt.show()
```

Listing B.15: Code for LUCAS: nutrients outlier count

```python
data = []

for nut in labels:
    Q1 = df[nut].quantile(0.25)
    Q3 = df[nut].quantile(0.75)
    # Calculate interquartile range (IQR)
    IQR = Q3 - Q1
    # Define lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Count outliers
    data.append(df[nut][(df[nut] < lower_bound) | (df[nut] > upper_bound)].
        count())

display(pd.DataFrame(data=data, columns=['# Outliers (IQR)'], index=labels))
```

Listing B.16: Code for LUCAS: histogram of ”LC0_Desc” classes

```python
# Gathering unique count data
df_data = df['LC0_Desc'].value_counts().reset_index()

plt.figure(figsize=(8, 8))
plot = sns.barplot(x="LC0_Desc", y="count", data=df_data)

for bar in plot.patches:
    # x-coordinate: bar.get_x() + bar.get_width() / 2
    # y-coordinate: bar.get_height()
    # free space to be left to make graph pleasing: (0, 8)
    # ha and va stand for the horizontal and vertical alignment
    plot.annotate(format(bar.get_height(), '.2f'),
                  (bar.get_x() + bar.get_width() / 2, bar.get_height()),
                  ha='center', va='center', size=15, xytext=(0, 8),
                      textcoords='offset points')

plt.xlabel("Land Cover 0", size=14)
plt.ylabel("Count", size=14)
plt.title("Land Cover Class frequency")
plt.show()
```

Listing B.17: Code for LUCAS: histogram of ”LC1_Desc” classes

```
# Gathering unique count data
df_data = df['LC1_Desc'].value_counts().reset_index()

# Defining the plot size
plt.figure(figsize=(8, 8))
# Defining the values for x-axis, y-axis
# and from which dataframe the values are to be picked
plot = sns.barplot(y="LC1_Desc", x="count", data=df_data)
plt.xlabel("Count", size=14)
plt.ylabel("Land Cover 1", size=14)
plt.title("Land Cover 1 frequency")
plt.show()
```

Listing B.18: Code for LUCAS: table with N, P and K data summary and their outliers according to the IQR method

```
data = []

# For each nutrient
for nut in labels:
    # Calculate quartiles
    Q1 = df[nut].quantile(0.25)
    Q3 = df[nut].quantile(0.75)
    # Calculate interquartile range (IQR)
    IQR = Q3 - Q1
    # Define lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Count outliers
    data.append(df[nut][(df[nut] < lower_bound) | (df[nut] > upper_bound)].
        count())

display(pd.DataFrame(data=data, columns=['# Outliers (IQR)'], index=labels))
```

```

df[labels].describe()
```

Listing B.19: Code for final dataset: Conversion to classification

```
# Fertility limits
N_fertility_limits = [10, 50]
P_fertility_limits = [10.9, 21.4]
K_fertility_limits = [40, 80]

# Auxiliar function for classification
def classify(x, lower, upper):
    if lower <= x <= upper:
        return 1.0
    elif x < lower or x > upper:
        return 0.0
    return x

# Classify each nutrient with function
for l, limits in zip(labels, (N_fertility_limits, P_fertility_limits,
        K_fertility_limits)):
    df[l] = df[l].apply(classify, args=limits)
```

Listing B.20: Code for final dataset: Balance assessment

```
data = []
N_total, N_fert, N_inf = df['N'].count(), df['N'][df['N'] == 0.0].count(), df
    ['N'][df['N'] == 1.0].count()
P_total, P_fert, P_inf = df['P'].count(), df['P'][df['P'] == 0.0].count(), df
    ['P'][df['P'] == 1.0].count()
K_total, K_fert, K_inf = df['K'].count(), df['K'][df['K'] == 0.0].count(), df
    ['K'][df['K'] == 1.0].count()

data.append((N_total, N_fert, N_fert/N_total, N_inf, N_inf/N_total))
data.append((P_total, P_fert, P_fert/P_total, P_inf, P_inf/P_total))
data.append((K_total, K_fert, K_fert/K_total, K_inf, K_inf/K_total))

```

```
pd.DataFrame(data=data, columns=('X_l', 'infertile', 'infertile (%)', '
    fertile', 'fertile (%)'), index=labels)
```

Listing B.21: Code for final dataset: Balance scatterplot

```
for l, name in zip(labels, ('Nitrogen', 'Phosphorus', 'Potassium')):
    values = df[l]
    if l == 'N':
        lower, upper = N_fertility_limits
    elif l == 'P':
        lower, upper = P_fertility_limits
    if l == 'K':
        lower, upper = K_fertility_limits
    # Define colors based on conditions
    # Define colors based on conditions
    colors = np.where((values < lower) | (values > upper), 'red', 'green')
    # Plot infertile points in red
    plt.scatter(np.where(colors == 'red')[0], values[colors == 'red'], color=
        'red', label=f'Infertile {name}')
    # Plot fertile points in green
    plt.scatter(np.where(colors == 'green')[0], values[colors == 'green'],
        color='green', label=f'Fertile {name}')

    plt.title(f'Distribution of {name} ({l})')
    plt.xlabel('Index')
    plt.ylabel(f'{l} (mg/kg)')
    # Add a legend and label for color interpretation
    plt.legend()
    plt.show()
```

Listing B.22: Code for final dataset: Trinary classification

```
# Auxiliar function for classification
def classify(x, lower, upper):
    if lower <= x <= upper:
        return 'sufficient'
    if x < lower:
        return 'defice'
    if x > upper:
        return 'toxic'
    return x

# Classify each nutrient with aux function
for l, limits in zip(labels, (N_fertility_limits, P_fertility_limits,
    K_fertility_limits)):
    df[l] = df[l].apply(classify, args=limits)
```

Listing B.23: Code for final dataset: Balance scatterplot for trinary P and K

```
for l, name in zip(('P', 'K'), ('Phosphorus', 'Potassium')):
    values = df[l]
    if l == 'P':
        lower, upper = P_fertility_limits
    elif l == 'K':
        lower, upper = K_fertility_limits
    # Define colors based on conditions
    defice = np.where(values < lower)
    fertile = np.where((values >= lower) & (values <= upper))
    toxic = np.where(values > upper)

    # Plot defice points in yellow
    plt.scatter(defice[0], values[values < lower], color='yellow', label=f'
        Defice')
    # Plot defice points in yellow
    plt.scatter(fertile[0], values[(values >= lower) & (values <= upper)],
        color='green', label=f'Fetile')
    # Plot defice points in yellow
    plt.scatter(toxic[0], values[values > upper], color='black', label=f'
        Toxic')

    plt.title(f'Distribution of {name} ({l})')
    plt.xlabel('Index')
    plt.ylabel(f'{l} (mg/kg)')
    # Add a legend and label for color interpretation
    plt.legend()
    plt.show()
```

Listing B.24: Code for final dataset: Balance table for trinary P and K

```
data = []

N_total, N_insuf, N_fert, N_tox = df['N'].count(), df['N'][df['N'] == 'sufficient'].count(), df['N'][df['N'] == 'defice'].count()\
    , df['N'][df['N'] == 'toxic'].count()
P_total, P_insuf, P_fert, P_tox = df['P'].count(), df['P'][df['P'] == 'sufficient'].count(), df['P'][df['P'] == 'defice'].count()\
    , df['P'][df['P'] == 'toxic'].count()
K_total, K_insuf, K_fert, K_tox = df['K'].count(), df['K'][df['K'] == 'sufficient'].count(), df['K'][df['K'] == 'defice'].count()\
    , df['K'][df['K'] == 'toxic'].count()

data.append((N_total, N_insuf, N_insuf/N_total, N_fert, N_fert/N_total, N_tox, N_tox/N_total))
data.append((P_total, P_insuf, P_insuf/P_total, P_fert, P_fert/P_total, P_tox, P_tox/P_total))
data.append((K_total, K_insuf, K_insuf/K_total, K_fert, K_fert/K_total, K_tox, K_tox/K_total))

pd.DataFrame(data=data, columns=('X_l',
                                  'defice', 'defice (%)',
                                  'fertile', 'fertile (%)',
                                  'toxic', 'toxic (%)'), index=labels)
```

Listing B.25: Prior code for tstk: get_bbox() and get_bbox_with_data()

```

    def get_bbox_with_data(self):
        """


        Returns:
            GeoDataFrame: Contains the bboxes which have associated groundtruth
        """
        if self._bbox_with_groundtruth is not None:
            return self._bbox_with_groundtruth
        self._bbox_with_groundtruth = self.get_groundtruth().copy(deep=True).
            reset_index()
        self._bbox_with_groundtruth.geometry = self.get_bbox().reset_index().
            geometry

        return self._bbox_with_groundtruth


    def get_bbox(self,buffer=0.005,reset=False):

        """
        Creates a grid of bbox over the dataset

        Args:
            dataset (GeoDataFrame): [description]
            expected_bbox_size (int, optional): The desired size of the bbox
                in meters. Defaults to 2000.
            reset (bool, optional): Wether it should recalculate the
                bbox_list. Defaults to False.

        Returns:
            GeoDataFrame: GeoDataFrame of the dataset divided in square bbox
                of size of expected_bbox_size.
        """

        #create bboxes around the groundtruth
        if self._dataset_bbox is not None and not reset:
            return self._dataset_bbox
        points_of_interest = [shapely.geometry.MultiPolygon([i.centroid.
            buffer(0.00001) for i in self.get_groundtruth().geometry.values])
            ]
        points_grid = self.get_groundtruth().geometry.apply(lambda x: BBox(x.
            centroid.buffer(buffer), sh.CRS.WGS84)).to_list()
        self.dataset_bbox_splitter = CustomGridSplitter(points_of_interest,
             sh.CRS.WGS84.pyproj_crs(),
```

```python
                points_grid)
        geometry = [Polygon(bbox.get_polygon()) for bbox in self.
            dataset_bbox_splitter.get_bbox_list()]
        self._dataset_bbox = gpd.GeoDataFrame(crs=sh.CRS.WGS84.pyproj_crs(),
            geometry=geometry)
        #self._dataset_bbox = self._dataset_bbox.drop_duplicates()
        return self._dataset_bbox
```

Listing B.26: Code for tstk: Downloader constructure

```python
# Creates dataset from shapefile
if shapefile is not None:
    self.dataset = gpd.read_file(shapefile)
    if groundtruth_col is not None:
        # Geometry keeps groundtruth geometry values
        has_geom = ~self.dataset[groundtruth_col].isna()
        self.dataset.loc[has_geom, 'geometry'] = loads(self.dataset.loc[
            has_geom, groundtruth_col])
        self.dataset.drop(columns=groundtruth_col, inplace=True)
```

Listing B.27: Code for tstk: Function to_square in utils.util.py

```python
def to_square(center_point: Point, area: int|float) -> Polygon:
    """
    #### Function that returns a Polygon of a square with center in Point\
    and with area in meters^2.

    Args:
        center_point: Point to use as the center of the square.
        area: Area in m2.

    Returns:
        Polygon with all corner points of square

    Raises ValueError and TypeError
    """
    if not isinstance(center_point, Point):
        raise TypeError(f'point must be Point. Got {type(center_point)}.')
    if type(area) is not int and type(area) is not float:
        raise TypeError(f'area must be int or float. Got {type(area)}.')

    l = sqrt(area)
    return center_point.buffer(l, cap_style='square')
```

Listing B.28: Code for tstk: Downloader.calculate_area()

```
def calculate_area(self, area: float|int = 2000) -> gpd.GeoDataFrame:
        """
        #### Function that substitutes each Point found in geometry\
        with a square Polygon with geom = 'area' and center in Point.

        Args:
            area: int or float equal to the area to be calculated.

        Return:
            GeoDataFrame

        Raises TypeError or ValueError
        """
        if type(area) is not int and type(area) is not float:
            raise TypeError(f'Area must be int or float. Got {type(area)}.')
        if area <= 0:
            raise ValueError(f'Area must be > 0. Got {area}.')
        self.dataset.to_crs('EPSG:3857', inplace=True)
        # apply function to all Point
        self.dataset['geometry'] = self.dataset['geometry'].apply(lambda g:
            to_square(g, area) if isinstance(g, Point) else g
        )
        self.dataset.to_crs(CRS.WGS84.pyproj_crs(), inplace=True)
        return self.dataset
```

Listing B.29: Code for tstk: Downloader.show_geometries()

```
def show_geometries(self) -> Map:
        """
        #### Function that returns global map with all geometries

        Returns:
            folium.Map
        """
        loc = self.dataset.unary_union.centroid.y, self.dataset.unary_union.
            centroid.x
        mapa = Map(location=loc, zoom_start=3)
        for g in self.dataset['geometry']:
            GeoJson(mapping(g)).add_to(mapa)
        return mapa
```

Listing B.30: Prior code for tstk: Downloader.download_images()

```
def download_images(self,path,bands=None,subset=None,date_field="SURVEY_DATE"
    ,date_fmt='%d/%m/%y',padding_days=2,maxcc=0.5):
        """Downloads the specified images into the users filesystem.

        Args:
            path (str): Path to where the dataset should be saved
            subset (DataFrame, optional): Slice of the dataframe returned by
                `get_bbox_with_data()`.
        """
        if bands is None:
            bands = ["B01","B02","B03","B04","B05","B06","B07","B08","B8A","
                B09","B11","B12"]
        if subset is None:
            subset = self.get_bbox_with_data()
        if not os.path.isdir(path):
            os.makedirs(path)

        add_data = SentinelHubInputTask(
            bands_feature=(FeatureType.DATA, 'BANDS'),
            resolution=10,
            bands=bands,
            maxcc=maxcc,
            time_difference=datetime.timedelta(minutes=120),
            data_collection=DataCollection.SENTINEL2_L2A,
            additional_data=[(FeatureType.MASK, 'dataMask'),
                     (FeatureType.MASK, 'CLM'),
                     (FeatureType.DATA, 'CLP')],
            max_threads=5
        )
        add_data_node = EONode(add_data,name="add_data_node")

        add_vector = AddFeatureTask((FeatureType.VECTOR_TIMELESS,"LOCATION"))
        add_vector_node = EONode(add_vector,[add_data_node],name="add_vector_node")
        #add_lucas = AddFeature((FeatureType.META_INFO,"LUCAS_DATA"))
        #to get the surrounding data, one can apply a buffered vector to raster and set the non overlapped value some value to distinguish
        add_raster_buffer = VectorToRasterTask((FeatureType.VECTOR_TIMELESS,"LOCATION"),(FeatureType.MASK_TIMELESS,"IS_VALID"), values = 5, buffer=0.0005, raster_shape=(FeatureType.MASK, 'CLM'), no_data_value=0,raster_dtype=np.uint8)
        add_raster_buffer_node = EONode(add_raster_buffer,[add_vector_node], name="add_raster_buffer_node")
        add_raster = VectorToRasterTask((FeatureType.VECTOR_TIMELESS,"LOCATION"),(FeatureType.MASK_TIMELESS,"IS_VALID"), values = 1, raster_shape=(FeatureType.MASK, 'CLM'),write_to_existing = True, no_data_value=0,raster_dtype=np.uint8)
        add_raster_node = EONode(add_raster,[add_raster_buffer_node],name="add_raster_node")

        concatenate = MergeFeatureTask({FeatureType.DATA: ['BANDS']},(FeatureType.DATA, 'FEATURES'))
        concatenate_node = EONode(concatenate,[add_raster_node],name="concatenate_node")
        save = SaveTask(path, overwrite_permission=OverwritePermission.OVERWRITE_PATCH)
        save_node = EONode(save,[concatenate_node],name="save_node")
        workflow = EOWorkflow([add_data_node,add_vector_node,add_raster_buffer_node,add_raster_node,concatenate_node,save_node])

```

```python
        execution_args = []
        for id, wrap_bbox in enumerate(subset.iterrows()):
            i, bbox = wrap_bbox

            time_interval = (get_time_interval(bbox[date_field],padding_days,
                date_fmt=date_fmt))
            gdf = gpd.GeoDataFrame(bbox)
            gdf = gpd.GeoDataFrame(gdf.transpose())
            gdf = gdf.rename(columns={0:'geometry'}).set_geometry('geometry')
            # gdf.set_geometry('geometry')
            gdf.crs = sh.CRS.WGS84.pyproj_crs()

            lucas_points_intersection = self.get_groundtruth()[self.
                get_groundtruth().geometry.values.intersects(gdf.geometry.
                values[0])]
            execution_args.append({
                add_vector_node:{'data': lucas_points_intersection},
                add_data_node:{'bbox': BBox(bbox.geometry, crs=self.dataset.
                    crs), 'time_interval': time_interval},
                save_node: {'eopatch_folder': f'eopatch_{id}'}
            })
        executor = EOExecutor(workflow, execution_args, save_logs=True)
        executor.run(workers=5, multiprocess=False)

        executor.make_report()
```

Listing B.31: Code for tstk: Downloader.download()

```python
def download(self, path: str, subset: gpd.GeoDataFrame = None,
                 date_fmt: str = '%d/%m/%y', padding_days: int = 2, maxcc:
                     float = 0.2, res: int = 10, report: bool = False, bands:
                      List[str] = None):
        """
        #### Downloads bands to EOPatches in 'path'.

        Args:
            path: Path to where the data is saved
            subset: Slice of the dataframe. Default is all of it
            date_frm: String representing the format of the date stored in df
            padding_days: Number of days subtracted and added to create the
                time interval
            maxcc: Percentage from 0-1 of cloud coverage allowed in data
                extracted
            res: Resolution of images. I.e. meters/pixel
            report: When true, saves the report as an html file
            bands: List of bands that user intends to extract. None is all
                available

        NOTE:
            - Bands are specified according to sentinel hub examples.
            - Entries where groundtruth_col is Nan will be computed according
                 to are.
        """
        if type(path) is not str:
            raise TypeError(f'path must be str. Got {type(path)}')
        if subset is not None and not isinstance(subset, gpd.GeoDataFrame):
            raise TypeError(f'subset must be geopandas.GeoDataFrame. Got {
                type(subset)}')
        else:
            subset = self.dataset
        if date_fmt is not None and type(date_fmt) is not str:
            raise TypeError(f'date_fmt must be str. Got {type(date_fmt)}')
        if padding_days is not None and type(padding_days) is not int:
            raise TypeError(f'padding_days must be int. Got {type(
                padding_days)}')
        if maxcc is not None and type(maxcc) is not float:
            raise TypeError(f'maxcc must be float. Got {type(maxcc)}')
        if res is not None and type(res) is not int:
            raise TypeError(f'res must be int. Got {type(res)}')
        if report is not None and type(report) is not bool:
            raise TypeError(f'report must be bool. Got {type(report)}')
        if bands is not None and type(bands) is not list:
            raise TypeError(f'bands must be list. Got {type(bands)}')

        # Parses requested bands
        if bands is None:
            bands = list(self.bands.values())
        else:
            for band in bands:
                if type(band) is not str:
                    raise TypeError(f'band must be str. Got {type(band)}')
            if not set(bands).issubset(self.bands.keys()):
                raise ValueError(f'Only {self.bands.keys()} bands are available. Got {bands}')
            bands = [b for b in self.bands.values() if b in bands]

        # Creates dir if not existing
        if not os.path.isdir(path):
            os.makedirs(path)

        # Node with task to extract all satelite bands
        data_task = SentinelHubInputTask(
            bands=bands,
            bands_feature=(FeatureType.DATA, "BANDS"),
            bands_dtype=float32,
            data_collection=self.collection,
            resolution=res,
            maxcc=maxcc,
            time_difference=datetime.timedelta(minutes=120),
            config=self.config
        )
        data_node = EONode(data_task, name='data_node')

        # Node with task to add nutrient values
        groundtruth_values_task = AddFeatureTask((FeatureType.META_INFO, 'GROUNDTRUTH'))
        groundtruth_values_node = EONode(groundtruth_values_task, [data_node],
                                         name="groundtruth_values_node")
        # Node with task to add date
        date_task = AddFeatureTask((FeatureType.META_INFO, 'DATE'))
        date_node = EONode(date_task, [groundtruth_values_node], name="date_node")

        # Node with task to save band names
        bands_names_task = AddFeatureTask((FeatureType.META_INFO, 'BAND_NAMES
            '))
        bands_names_node = EONode(bands_names_task, [date_node], name='
            bands_names_node')

        # Node with task that saves the EOPatches != None into path
        save = SaveNotNullTask(path, save_timestamps=True,
                                 overwrite_permission=OverwritePermission.
                                     OVERWRITE_FEATURES)
        save_node = EONode(save, [bands_names_node], name="save_node")

        # Creates sequence of EONodes to be executed
        workflow = EOWorkflow([data_node, groundtruth_values_node, date_node,
                                 bands_names_node, save_node])

        execution_args = []
        for index, row in subset.iterrows():
            # Gets the id
            id = self.id if self.id is not None else index
            # Calculates time interval
            time_interval = get_time_interval(row['date'], padding_days,
                date_fmt=date_fmt)

            bbox = BBox(row['geometry'].bounds, crs=self.dataset.crs)
            # Creates parameters for each EONode execution
            execution_args.append({
                data_node: {'bbox': bbox, 'time_interval': time_interval},
                groundtruth_values_node: {'data': row[self.keep].to_dict()},
                date_node: {'data': row['date']},
                bands_names_node: {'data': self.bands},
                save_node: {'eopatch_folder': f'eopatch_{row[id]}'}
            })

        # Executes workflow
        executor = EOExecutor(workflow, execution_args, save_logs=report)
        executor.run()
        # Saves report
        if report:
            executor.make_report()
```

Listing B.32: Code for tstk: SaveNotNullTask() in utils.eotasks.py

```
class SaveNotNullTask(SaveTask):
    """
    #### Only saves EOPatches that have band data.
    """
    def execute(self, eopatch, **kwargs):
        if eopatch[(FeatureType.DATA, 'BANDS')].size > 0:
            super().execute(eopatch, **kwargs)
```

Listing B.33: Code for tstk: TSPatch() constructor

```
class TSPatch(EOPatch):
    def __init__(self, id: str, eopatch: EOPatch = None):
        """
        #### Extends the functionality of the original eo-patch
            implementation with new methods

        Args:
            id: String used to identify the TSPatch
            EOPatch: eo-learn abstraction to represent a single region
        """
        if eopatch is not None:
            self.eopatch = eopatch
        else:
            super().__init__()
            self.eopatch = super()
        # Path for TSPatch in sys
        self.id = id
        # Saves location of EOPatch
        self.location = self.bbox.geometry.centroid.x, self.bbox.geometry.
            centroid.y
        # Saves available bands
        self.bands, self.indices = {}, []
```

```python
        for v, b in enumerate(eopatch.meta_info['BAND_NAMES']):
            self.bands[b] = v
```

Listing B.34: Code for tstk: TSPatch.add_indice()

```python
def add_indice(self, indice: str, arr: np.ndarray):
        """
        #### Adds indice to eopatch

        Args:
            indice: name of indice to add
            arr: np array with value to store
        """
        if type(indice) is not str:
            raise TypeError(f'{indice} must be str. Got {type(indice)}')
        if not isinstance(arr, np.ndarray):
            raise TypeError(f'{arr} must be np.ndarray. Got {type(arr)}')
        if np.ndim(arr) != 4:
            raise ValueError(f'{arr} must have 4 dimentions (t, m, n, i). Got
                 {arr.shape}')
        if self.eopatch.__contains__((FeatureType.DATA, indice)):
            raise ValueError(f'Indice {indice} already exists.')

        self.data[indice] = arr
        self.indices.append(indice)
```

Listing B.35: Code for tstk: TSPatch.get_location()

```python
def get_location(self) -> Tuple[float, float]:
        """
        #### Getter for gps location of TSPatch

        Returns:
            Tuple: With latitude and longitude values
        """
        return self.location
```

Listing B.36: Prior code for tstk: TSPatch.get_masked_region()

```python
def get_masked_region(self):
        mask = self.patch.mask_timeless["IS_VALID"].squeeze()
        mask_filtered = np.where(mask==5,0,mask)
        return mask_filtered
```

Listing B.37: Code for tstk: TSPatch.get_masked_region()

```python
def get_masked_region(self, indice: str = None) -> np.ndarray:
        """
        #### Function that returns a numpy.array marking where data is valid
        #### When EOPatch has no mask info, all points are valid
        NOTE: 1 means valid. 0 means not valid

        Args:
            indice: String of indice to return array with it's shape. Default
                returns the bands mask

        Returns:
            numpy.ndarray: Mask to apply to patches, where 1 is valid
        """
        # All points are valid if there is no mask
        if not self.eopatch.__contains__(FeatureType.MASK_TIMELESS):
            if indice is not None:
                # Checks for indice existance
                if not self.eopatch.__contains__((FeatureType.DATA, indice)):
                    raise ValueError(f'Indice {indice} does not exist. Only {
                        self.eopatch.get_features()}')

                # Returns with indice's shape full of 1's
                return np.ones_like(self.data[indice][0])

            # Returns array like BANDS full of 1's
            return np.ones_like(self.data['BANDS'][0])

        mask = self.mask_timeless["IS_VALID"].squeeze()
        mask_filtered = np.where(mask==5, 0, mask)
        return mask_filtered
```

Listing B.38: Prior code for tstk: TSPatch.get_masked_region_values()

```
def get_values_of_masked_region(self ,indices = None ,band_names = None,
    as_array=True):
        """Returns the pixels in the masked region for the selected `indices`
            and `band_names` for each of the patch

        Args:
            indices (str): The list of indices in which we want to get the
                values of
            band_names (str): The list of indices in which we want to get the
                values of
            as_array (bool, optional): If true returns in 1D array form(only
                the values with data), else returns in 2D array. Defaults to
                True.

        Returns:
            ndarray: If `as_array` is true returns in 1D array form(only the
                values with data), else returns in 2D array.
        """
        values = {}
        eopatch = self.patch
        masked_region = self.get_masked_region()
        nearest_image_index = self._get_index_nearest_to_collection_date()
        for i in range(0,eopatch.data["BANDS"].shape[-1]):
            values[band_names[i]] = eopatch.data["BANDS"][nearest_image_index
                ,...,i]*masked_region
            if as_array:
                values[band_names[i]] = values[band_names[i]][masked_region!=
                    0]

        for i in indices:
            values[i] = eopatch.data[i][nearest_image_index,...,-1]*
                masked_region
            if as_array:
                values[i] = values[i][masked_region!=0]
        return values
```

Listing B.39: Code for tstk: TSPatch.get_masked_region_values()

```
def get_values_of_masked_region(self, indices: List[str] = None, bands: List[str] = None,
                                as_array: bool = True, verbose: bool = False
                                ) -> dict[str: np.ndarray]:
        """
        #### Returns the pixels in the masked region for the selected `indices` and `bands`\
        for each of the patch

        Args:
            indices: The list of indices in which we want to get the values of.
                     None means all indices.
            bands: The list of bands names of which we want to get the values of.
                   None means all bands.
            as_array: If true returns in 1D array form(only the values with data),
                      else returns in 2D array. Defaults to True.
            verbose: boolean to print important runtime information.

        Returns:
            dict: Dictionary mapping indice/band name to numpy.ndarray with values
        """
        result = {}

        # Gets masked region for indices
        if indices is not None:
            # Indices must be list
            if type(indices) is not list:
                raise TypeError(f'Indices must be list. Got {type(indices)}')
            # All in indices must be str
```

```python
            if not all(type(val) is str for val in indices):
                raise TypeError(f'Indices must be list of strings. Got {type(indices[0])}')
            # Check if indices exist in eopatch
            if not set(indices).issubset(self.indices):
                raise ValueError(f'Indice must exist. Available are: {self.indices}')

        indices = self.indices if indices is None else indices

        # Parser input band names
        if bands is not None:
            # Bands must be list
            if type(bands) is not list:
                raise TypeError(f'Bands must be list. Got {type(bands)}')
            # All in bands must be str
            strings = [type(b) is str for b in bands]
            if not all(strings):
                wrong_idx = strings.index(False)
                raise TypeError(f'Bands must be list of strings. Got {type(bands[wrong_idx])}')
            # Check if bands exist in eopatch
            if not set(bands).issubset(self.bands.keys()):
                raise ValueError(f'Band must exist. Available are: {self.bands.keys()}')

        bands = list(self.bands.keys()) if bands is None else bands

        # Closest to date index
        closest_i = self._get_index_nearest_to_collection_date(verbose)

        if len(indices) > 0:
            # Gets mask with indice shapelike
            indices_mask = self.get_masked_region(indices[0])
            # Calculate mask for indices
            for indice in indices:
                result[indice] = self.data[indice][closest_i] * indices_mask
                if as_array:
                    result[indice] = result[indice].flatten()

        # Gets masked region for bands
        for band in bands:
            idx = self.bands[band]
            # Gets mask only for current band
            band_mask = self.get_masked_region()[..., idx]
            result[band] = self.data['BANDS'][closest_i][..., idx] *
                band_mask
            if as_array:
                result[band] = result[band].flatten()

        return result
```

Listing B.40: Prior code for tstk: TSPatch.represent_image()

```
def represent_image(self,estimation):
        """Draws an image with the values estimated

        Args:
            estimation (array): Array with the size of the masked region(1D)

        Returns:
            array: 2D image
        """
        nearest_image_index = self._get_index_nearest_to_collection_date()

        eopatch = self.patch
        mask = self.get_masked_region()
        mask = mask.astype(float)
        image = eopatch.data["BANDS"][nearest_image_index][...,[3,2,1]]
        _max = 255#dfeopatches["N"].max()
        _min = 0#dfeopatches["N"].min()
        convert_est = estimation/_max
        for i,tup in enumerate(zip(mask.nonzero()[0],mask.nonzero()[1])):
            x = tup[0]
            y = tup[1]
            try:
```

```python
                image[x,y] = [convert_est[i], 0,0]
            except BaseException as err:
                print(f"The estimation most likely doesn't correspond to the eopatch: {err}")
        return image
```

Listing B.41: Code for tstk: TSPatch.represent_image()

```python
def represent_image(self, R: str, G: str, B: str, factor: int = 255, verbose: bool = False
                        ) -> np.ndarray:
        """
        #### Returns an image array in RGB with values from features mentioned.

        Args:
            R: Name of indice/band used for Red axis.
            G: Name of indice/band used for Green axis.
            B: Name of indice/band used for Blue axis.
            factor: Float to multiply with values
            verbose: boolean to print important runtime information.

        Returns:
            array: 2D image
        """
        # Gets the index of the patch taken closer to the rewuested time
        nearest_indx = self._get_index_nearest_to_collection_date(verbose)

        # Saves bands shape
        bands_shape = self.data['BANDS'].shape
        # Image is line, column and [R, G, B]
        img = np.ones(shape=(bands_shape[1], bands_shape[2], 3))

        # Check arguments
        for n, axis in enumerate((R, G, B)):
            # Axis is a prev calculated indice
            if type(axis) is str:
                if axis not in self.indices and axis not in self.bands.keys()
```

```
                    :
                    raise ValueError(f'Parameter does not exist. Available: {self.indices} or {list(self.bands.keys())}')

                if axis in self.bands.keys():
                    # Get Band values
                    img[..., n] = self.data['BANDS'][nearest_indx][..., self.bands[axis]] * factor
                else:
                    # Get indice values
                    img[..., n] = self.data[axis][nearest_indx][..., 0] * factor

            else:
                raise TypeError(f'Parameters must be str. Got: {type(R), type(G), type(B)}')

        return img
```

Listing B.42: Prior code for tstk: TSPatch._get_index_nearest_to_collection_date()

```
def _get_index_nearest_to_collection_date(self):
        smallest_index = 0
        smallest_difference = dt.timedelta(days=2000)
        try:
            collected_day = dt.datetime.strptime(self.get_dataset_entry_value("SURVEY_DATE"),'%d/%m/%y')
        except:
            return -1
        for i,image_date in enumerate(self.timestamp):
            current_difference = abs(collected_day - image_date)
            if(current_difference < smallest_difference):
                smallest_difference = current_difference
                smallest_index = i
        return smallest_index
```

Listing B.43: Code for tstk: TSPatch._get_index_nearest_to_collection_date()

```python
def _get_index_nearest_to_collection_date(self, verbose: bool = False):
        """
        This function gets the index of the temporal band closest to the collection data that is not\
        full of 0's.

        Args:
            verbose: boolean to print important runtime information.

        Raises ValueError if all temporal bands are full of 0's
        """
        collected_day = datetime.strptime(self.meta_info['date'], '%d/%m/%Y')

        if len(self.timestamp) == 1:
            result = 0
        else:
            sorted_timestamps = sorted(self.timestamp, key=lambda t: abs(collected_day - t))
            result = self.timestamp.index(sorted_timestamps[0])

        # Check if array is full of zeros across all dimensions
        if np.all(self.data['BANDS'][result] == 0, axis=None):
            if verbose:
                print('INFO: EOP nearest to collection date is full of 0.')
            if len(self.timestamp) == 1:
                raise ValueError(f'INFO: No more EOP in TSP, thus ignoring this {self.id}.')

            if verbose:
                print('INFO: Getting 2nd closest and non-zero EOP.')
            for t in range(1, len(sorted_timestamps)):
                indx = self.timestamp.index(sorted_timestamps[t])
                if np.all(self.data['BANDS'][indx] != 0, axis=None):
                    if verbose:
                        print(f'INFO: Got EOP with {abs(collected_day - sorted_timestamps[t])} days different.')
                    return indx

            raise ValueError(f'INFO: No more EOP in TSP, thus ignoring this {
                self.id}.')

        return result
```

Listing B.44: Code for tstk: Dataset() and Dataset.__create_append_tsp__()

```
class Dataset:
    """
    #### Class that represents a collection of EOPatches saved in disk

    Args:
        eops_path: String with path to directory with EOPatches to be read
    """
    def __init__(self, eops_path: str):
        # Checks if path exists
        if type(eops_path) is not str:
            raise TypeError(f'Path must be str.')
        if not isdir(eops_path):
            raise ValueError(f'Path given: {eops_path} is not directory.')

        # Path to directory with eopatches
        self.eops_path = eops_path

        # Parallelize TSP creation
        self.tspatches = []
        self.lock = Lock()
        list_dir = listdir(eops_path)
        with concurrent.futures.ThreadPoolExecutor() as executor:
            executor.map(self.__create_append_tsp__, list_dir)

        # Pandas dataframe with info about TSPatches groundtruth data
        self.gth_df = None
        # Saves available indices in EOPatches
        self.indices = []
        # Saves dict of available bands in EOPatches
        self.bands = self.tspatches[0].get_bands()
        # Saves groundtruth data names
        self.groundtruth_f = self.tspatches[0].get_groundtruth_features()


    # Method to parallelize TSP creation
    def __create_append_tsp__(self, eop_dir):
        full_path = join(self.eops_path, eop_dir)
        tsp = TSPatch.load(full_path, lazy_loading=True)
        with self.lock:
            self.tspatches.append(tsp)
```

Listing B.45: Code for tstk: Dataset.add_indices()

```
def add_indices(self, indice_func: dict[str: Callable], save: bool = False,
    report: bool = False,
                    verbose: bool = False, override_values: bool = False):
        """
        #### Function that receives a mapping for indice's names to how they
            are calculated, \
        and adds them to list of TSPatches

        Args:
            indice_func: Dictionary mapping indice name and how it is
                calculated.
            save: Boolean to overwrite data and save indice to EOPatch in sys
                .
            report: Boolean specifies if user wants execution report.
            verbos: Boolean to show important runtime information.
            override_values: When True, tstk changes the band values that
                give mathematical errors.

        Example:
            >>> bands = dataset.get_bands()
            >>> indices = {'EVI2': lambda b: 2.4*(b[bands['B:NIR']] - b[bands
                ['B:GREEN']]}
            >>> dataset.add_indices(indices)
            >>> dataset.add_indices(indices, save=True, report=True)

        NOTE: All temporal bands full of 0 are ignored. Moreover, if valid temporal band\
            has 0 values, they are incremented by 1e-3 to avoid mat errors.
        """
        np.seterr(all='raise')

        # Node to save new indice in
        if save:
            save_task = SaveTask(self.eops_path, features=[(FeatureType.DATA, indice)])
            save_node = EONode(save_task, name=f'save_{indice}_node')
            workflow, exec_args = EOWorkflow([save_node]), []

        for tsp in self.get_tspatches():
            eop_bands = tsp.data['BANDS']

            # invalid temporal bands <=> bands with nothing but 0
            inv_t_b = np.all(eop_bands==0, axis=(1, 2, 3))
            # Some or all temporal bands are full of 0
            if np.sum(inv_t_b) > 0:
                if verbose:
                    print(f'INFO: TSP {tsp.id} has {np.sum(inv_t_b)} temporal bands full of 0. These are ignored and kept as 0.')
                # All temporal bands are invalid. Skipping to next tsp
                if np.sum(inv_t_b) == eop_bands.shape[0]:
                    if verbose:
                        print(f'INFO: Skipping TSP {tsp.id} calculation. All temporal bands are 0.')
                    continue

            # Increments random small value on all valid temporal bands to avoid calculation errors
            if np.sum(eop_bands[~inv_t_b] == 0) > 0:
                if verbose:
                    print(f'INFO: TSP {tsp.id} has {np.sum(eop_bands[~inv_t_b] == 0)} pixels = 0 in valid temporal bands. These are incremented by 1e-3.')
                zero_indices = np.where(eop_bands[~inv_t_b] == 0)
```

```python
                eop_bands[zero_indices] += np.random.random(size=len(zero_indices[0])) * 1e-3

            for indice, func in indice_func.items():

                if tsp.__contains__((FeatureType.DATA, indice)):
                    if verbose:
                        print(f'INFO: TSP {tsp.id} already has indice {indice}. Skipping calculation.')
                    continue

                indice_eop = np.zeros(shape=eop_bands.shape[:-1], dtype=np.float32)
                # Apply function. func is applied to each 3rd dimetion = Each pixel
                try:
                    indice_eop[~inv_t_b] = np.apply_along_axis(func, axis=3, arr=eop_bands[~inv_t_b])
                except FloatingPointError:
                    if verbose:
                        print(f'SVI {indice} is mathematically impossible to calculate on TSP {tsp.id}')

                # Bands temporal is (n, m) but to store as DATA it needs to be (n, m, b)
                # even though it should be 3d, storing as DATA_TIMELESS seems incorrect
                # So we add an unecessary dimention
                indice_eop = np.expand_dims(indice_eop, axis=-1)
                tsp.add_indice(indice, indice_eop)

            if save:
                exec_args.append({save_node: {'eopatch_folder': tsp.id, 'eopatch': tsp.eopatch}})

        # Save in sys
        if save:
            executor = EOExecutor(workflow, exec_args, save_logs=report)
            executor.run(multiprocess=False)
            # Saves report
            if report:
                executor.make_report()

        for i in indice_func.keys():
            if i not in self.indices:
                self.indices.append(i)
```

Listing B.46: Prior code for tstk: Dataset.add_index() and Dataset.save_indices_to_eopatches()

```
def add_index(self,index_name, index_formula):
        """Adds a specific index to the dataset in question

        NOTE:To make the index persist `save_indices_to_patches` must be
            called

        Args:
            index_name (str): The name of the index
            index_formula (str): The formula for the specified index

        Example:

            >>> dataset.add_index("NDVI","(B07-B04)/(B07+B04)")
            >>> dataset.add_index("IRECI","(B07-B04)/(B05/B06)")
            >>> dataset.save_indices_to_patches()

        """
        self.index_dic[index_name] = index_formula



    def save_indices_to_eopatches(self):
        """Saves the previously indicated indices into the dataset.

        Example:

            >>> dataset.add_index("NDVI","(B07-B04)/(B07+B04)")
            >>> dataset.add_index("IRECI","(B07-B04)/(B05/B06)")
            >>> dataset.save_indices_to_patches()
        """
        load = LoadTask(self.eopatches_folder)
        load_node = EONode(load,name="Load")
        available_bands = ['B01', 'B02', 'B03', 'B04', 'B05', 'B06', 'B07', '
            B08', 'B8A', 'B09', 'B10', 'B11', 'B12']

        add_indices = AddIndicesTask(self.index_dic,available_bands)
        add_indices_node = EONode(add_indices,[load_node],name="Add_index")
        save = SaveTask(self.eopatches_folder, overwrite_permission=
            OverwritePermission.OVERWRITE_PATCH)
        add_save_node = EONode(save,[add_indices_node],name="Save_patch")
        execution_args = []
        workflow = EOWorkflow([load_node, add_indices_node,add_save_node])
        eopatch_folders = os.listdir(self.eopatches_folder)
        print(len(eopatch_folders))
        for i in eopatch_folders:
            execution_args.append(
            {
                load: {'eopatch_folder': f'{i}'},
                add_indices: {},
                save: {'eopatch_folder': f'{i}'}
            })
        executor = EOExecutor(workflow, execution_args, save_logs=True)
        executor.run(workers=5, multiprocess=False)
        executor.make_report()
```

Listing B.47: Code for tstk: Dataset.add_indices()

```
def add_indices(self, indice_func: dict[str: Callable], save: bool = False,
    report: bool = False,
                    verbose: bool = False, override_values: bool = False):
        """
        #### Function that receives a mapping for indice's names to how they
            are calculated, \
        and adds them to list of TSPatches

        Args:
            indice_func: Dictionary mapping indice name and how it is calculated.
            save: Boolean to overwrite data and save indice to EOPatch in sys.
            report: Boolean specifies if user wants execution report.
            verbos: Boolean to show important runtime information.
            override_values: When True, tstk changes the band values that give mathematical errors.

        Example:
            >>> bands = dataset.get_bands()
            >>> indices = {'EVI2': lambda b: 2.4*(b[bands['B:NIR']] - b[bands['B:GREEN']]}
            >>> dataset.add_indices(indices)
            >>> dataset.add_indices(indices, save=True, report=True)

        NOTE: All temporal bands full of 0 are ignored. Moreover, if valid temporal band\
            has 0 values, they are incremented by 1e-3 to avoid mat errors.
        """
        np.seterr(all='raise')

        # Node to save new indice in
        if save:
            save_task = SaveTask(self.eops_path, features=[(FeatureType.DATA, indice)])
            save_node = EONode(save_task, name=f'save_{indice}_node')
            workflow, exec_args = EOWorkflow([save_node]), []

        for tsp in self.get_tspatches():
            eop_bands = tsp.data['BANDS']

            # invalid temporal bands <=> bands with nothing but 0
            inv_t_b = np.all(eop_bands==0, axis=(1, 2, 3))
            # Some or all temporal bands are full of 0
            if np.sum(inv_t_b) > 0:
                if verbose:
```

```python
                    print(f'INFO: TSP {tsp.id} has {np.sum(inv_t_b)} temporal bands full of 0. These are ignored and kept as 0.')
                # All temporal bands are invalid. Skipping to next tsp
                if np.sum(inv_t_b) == eop_bands.shape[0]:
                    if verbose:
                        print(f'INFO: Skipping TSP {tsp.id} calculation. All temporal bands are 0.')
                    continue

            # Increments random small value on all valid temporal bands to avoid calculation errors
            if np.sum(eop_bands[~inv_t_b] == 0) > 0:
                if verbose:
                    print(f'INFO: TSP {tsp.id} has {np.sum(eop_bands[~inv_t_b] == 0)} pixels = 0 in valid temporal bands. These are incremented by 1e-3.')
                zero_indices = np.where(eop_bands[~inv_t_b] == 0)
                eop_bands[zero_indices] += np.random.random(size=len(zero_indices[0])) * 1e-3

            for indice, func in indice_func.items():

                if tsp.__contains__((FeatureType.DATA, indice)):
                    if verbose:
                        print(f'INFO: TSP {tsp.id} already has indice {indice}. Skipping calculation.')
                    continue

                indice_eop = np.zeros(shape=eop_bands.shape[:-1], dtype=np.float32)
                # Apply function. func is applied to each 3rd dimetion = Each pixel
                try:
                    indice_eop[~inv_t_b] = np.apply_along_axis(func, axis=3, arr=eop_bands[~inv_t_b])
                except FloatingPointError:
                    if verbose:
                        print(f'SVI {indice} is mathematically impossible to
calculate on TSP {tsp.id}')

                # Bands temporal is (n, m) but to store as DATA it needs to be (n, m, b)
                # even though it should be 3d, storing as DATA_TIMELESS seems incorrect
                # So we add an unecessary dimention
                indice_eop = np.expand_dims(indice_eop, axis=-1)
                tsp.add_indice(indice, indice_eop)

            if save:
                exec_args.append({save_node: {'eopatch_folder': tsp.id, 'eopatch': tsp.eopatch}})

        # Save in sys
        if save:
            executor = EOExecutor(workflow, exec_args, save_logs=report)
            executor.run(multiprocess=False)
            # Saves report
            if report:
                executor.make_report()

        for i in indice_func.keys():
            if i not in self.indices:
                self.indices.append(i)
```