Sen4CAP - Sentinels for Common Agricultural Policy

Design Definition File

ATBD for L4B grassland mowing detection product

Milestone	Milestone 1
Authors	Laura DE VENDICTIS, Cecilia SCIARETTA, Florin TUTUNARU, Massimo ZAVAGLI, Sophie BONTEMPS
Distribution	ESA - Benjamin KOETZ

This page is intentionally left blank

Table of contents

List of figures

List of tables

List of acronyms

Acronym	Definition
CFAR	Constant false alarm rate
CZE	Czech Republic
ESP	Spain
FAPAR	Fraction of Absorbed Photosynthetically Active Radiation
FAR	False Positive Rate
FID	Feature Identifier
GSAA	GeoSpatial Aid Application
ITA	Italy
LAEA	Lambertian Azimuthal equal-area
LAI	Leaf Area Index
LPIS	Land Parcel Identification System
LTU	Lithuania
NDVI	Normalized Difference Vegetation Index
NLD	Netherlands
ROU	Romania
SAR	Synthetic Aperture Radar
S-1	Sentinel-1
S-2	Sentinel-2
VI	Vegetation Index

1. Logical model

1.1 Overview of the system

The grassland mowing detection product (L4B) is based on the processing of Sentinel 1 (S-1) and Sentinel-2 (S-2) derived products, which are respectively coherences, calibrated amplitude backscatter (square root of σ_0 (sigma_nought)) and three vegetation indicators (VIs) which are the Normalized Difference Vegetation Index (NDVI), the Fraction-Absorbed Photosynthetically Active Radiation (FAPAR) and the Leaf Area Index (LAI). The processing is performed independently on the Synthetic Aperture Radar (SAR) and on the multispectral optical data and the results are merged to provide a single output by fusing the detections through the exploitation of their reliability indicators. Following this approach, the system architecture is mainly based on two parallel processing chains linked at the final stage through the "mowing detections fusing" module.

The internal architecture of the system is depicted in Figure 1-1. The elements of such system are:

- S-1 data processing units:
  - SAR Feature extraction
  - SAR mowing detection
- S-2 data processing units
  - VI Feature extraction
  - VI mowing detection
- Common processing units
  - Fusion of detections and compliancy assessment

The external interface of the L4B processing system (red dashed lines in Figure 1-1) are:

- S-1 derived product data (input amplitudes VV, VH and coherences VV, VH):
  - Formats
  - Directory structure
- S-2 derived product data (input NDVI, LAI, FAPAR):
  - Formats
  - Directory structure
- Ancillary data (input):
  - Formats
  - Directory structure
- Result (output):
  - Format
  - Directory structure

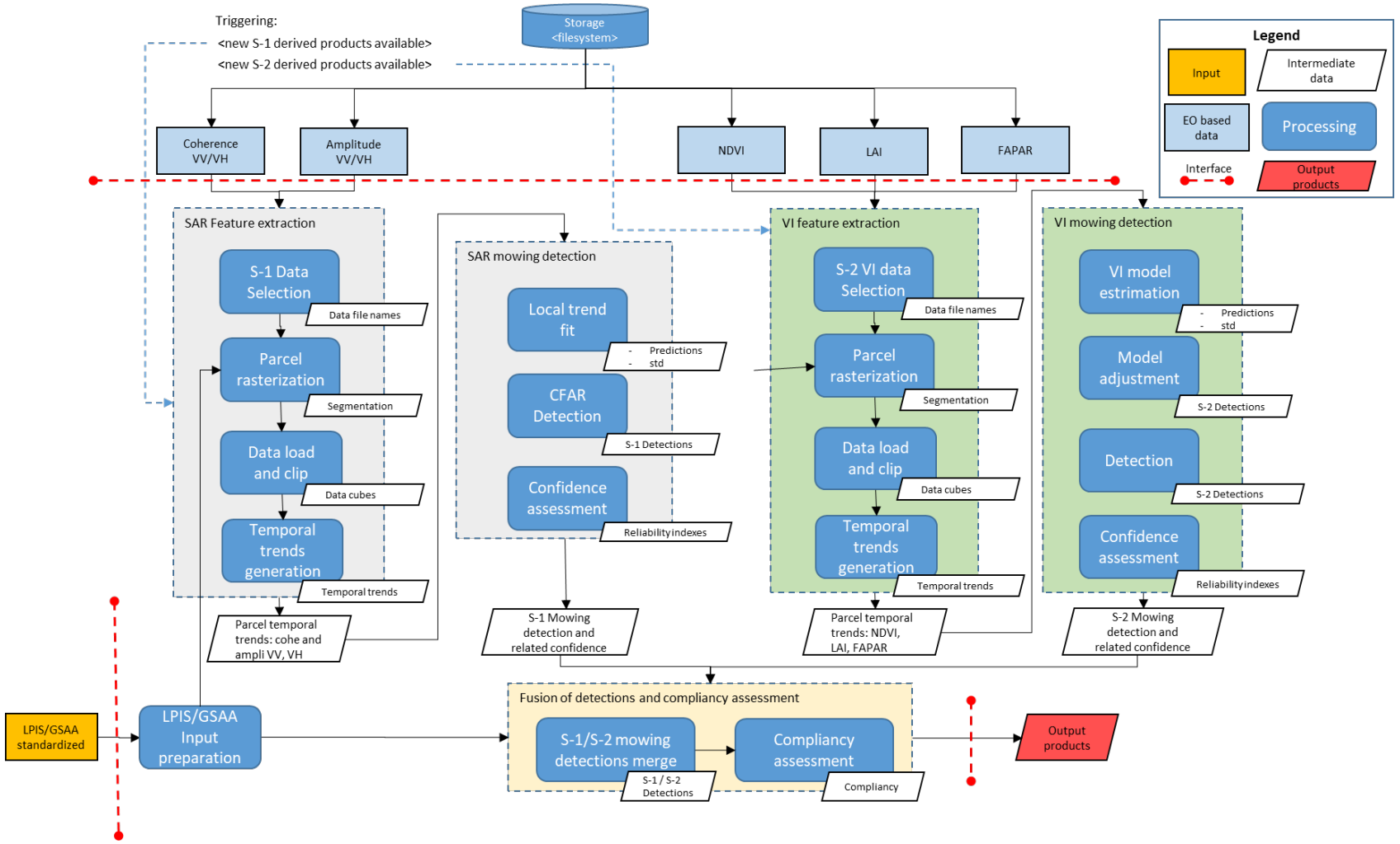

Figure 1-1. System architecture

2. L4B system components

2.1 External interface of the L4B processing system

2.1.1 Input data and parameters

The mowing detection is performed on S-1 and S-2 derived products, respectivelly calibrated amplitude backscatterer (square root of σ_0) and short-term (6 days) coherences for S-1 and vegetation indexes for S-2 data. This section describes the derived products, ancillary data and source directory structure and naming conventions as they are currently used:

- S-1 pre-processed data :
  - calibrated amplitude backscatterer (square root of σ_0) images VV and VH:
    - Type: geotiff raster images, float 32bit;
      Naming convention:
      SEN4CAP_L2A_PRD_Sn_yyyyMMddThhmmss_VyyyyMMddThhmmss_yyyyMMddThhmmss_PP_RRR_AMP.tif (.nc)
      where Sn = site identifier (n=1,2,3,4,5,6), _yyyyMMddThhmmss_ = product creation timestamp, VyyyyMMddThhmmss_yyyyMMddThhmmss= V<master acquisition date and time>_<slave acquisition date and time>, PP = polarization (VV or VH), RRR = relative orbit.
  - Short term (6-days) Coherences VV and VH:
    - Type: geotiff raster images, float 32bit;
    - Naming convention:
      SEN4CAP_L2A_PRD_Sn_yyyyMMddThhmmss_VyyyyMMddThhmmss_yyyyMMddThhmmss_PP_RRR_COHE.tif (.nc)
      where Sn = site identifier (n=1,2,3,4,5,6), _yyyyMMddThhmmss_ = product creation timestamp, VyyyyMMddThhmmss_yyyyMMddThhmmss= V<master acquisition date and time>_<slave acquisition date and time>, PP = polarization (VV or VH), RRR = relative orbit.

- S-2 pre-processed data :
  - Vegetation Indexes (FAPAR, NDVI, LAI) :
    - Type: geotiff raster images, integer 32bit;
    - Naming convention:
      S2AGRI_L3B_<VIType>_A<yyyymmdd>T<hhmmss>_<TileID>.TIF;
      where <yyyymmdd> is the 8-digits acquisition date, <hhmmss> is the 6-digit time, <VIType> = [‘SNDVI’, ‘SFAPARMONO’, ‘SLAIMONO’], <TileID> is the 6-digit tile identifier.
  - Source directory structure :
    - Vegetation Indexes (FAPAR, NDVI, LAI): <S2DataRoot>/ S2AGRI_L3B_PRD_S2_<YYYYMMDD>T<HHMMSS>_A<yyyymmdd>T<hhmmss>/TILES/S2AGRI_L3B_A<yyyymmdd>T<hhmmss>_<TileID>/IMG_DATA/ ; where <yyyymmdd> is the 8-digits acquisition date, <hhmmss> is the 6-digit acqusition time, <YYYYMMDD> is the 8-digits preprocessing date, <HHMMSS> is the 6-digit preprocessing time, <VIType> = [‘SNDVI’, ‘SFAPARMONO’, ‘SLAIMONO’], <TileID> is the 6-digit tile identifier.
  - Source directory structure for GSAA input data: None

The input variables required to access and read the input S-1 and S-2 derived products and GSAA input data are depicted in Table 2-1.

Table 2-1. Input variables to access the input S-1 and S-2 derived products and GSAA input data data

Input variable	Role	Default value
SARDataRoot	String with the root directory of the source directory structure of the S-1 preprocessed data	NA
S2DataRoot	String with the root directory of the source directory structure of the S-2 preprocessed data	NA
SegmentsFile	String with the file name with absolute path of the shape file. (e.g. /var/scratch/LPIS/LPIS.shp)	NA
seg_parcel_id_attribute	String with attribute name corresponding to the parcel IDs.	'parcel_id'

Once the data are ingested and loaded, the following data structures are generated in memory to be passed to the following modules SAR data processing and Optical data processing.

2.1.2 Output data and parameters

The output of the mowing detection process is stored in an output shapefile file <outputShapeFile>. This file must be available at the start of the processing and must have the same structure of the input file <inputShapeFile>:

[1] NewID: Unique Parcel ID (int)
[2] Ori_hold: Holding ID (string)
[3] Ori_id: Original Parcel ID (string)
[4] Ori_crop: Code of crop type selected as grassland (string)
[5] mow_n: number of the detected mowings events (int in [0,1,2,3,4])
[6] m1_dstart: date and time of the start mowing (string)
[7] m1_dend: date and time of the end mowing (string)
[8] m1_conf: confidence level (float)
[9] m1_mis: satellite mission (string)
[10] m2_dstart: date and time of the start mowing (string)
[11] m2_dend: date and time of the end mowing (string)
[12] m2_conf: confidence level (float)
[13] m2_mis: satellite mission (string)
[14] m3_dstart: date and time of the start mowing (string)
[15] m3_dend: date and time of the end mowing (string)
[16] m3_conf: confidence level (float)
[17] m3_mis: satellite mission (string)
[18] m4_dstart: date and time of the start mowing (string)
[19] m4_dend: date and time of the end mowing (string)
[20] m4_conf: confidence level (float)
[21] m4_mis: satellite mission (string)
[22] compl (int in [0,1,2])

The same output shapefile file (Table 2-2) is used to store the results of both the S-1 and S-2 based processing chains and it is updated iteratively at each processing run, in the attributes from [5] to [22].

Table 2-2. Output of the mowing detection process

Output parameters	Role	Default value
outputShapeFile	String with the file name with absolute path of the output shapefile file. (e.g. /var/scratch/output/output_file.shp)	NA

2.2 GSAA input shapefile preparation

This part of the processing receives as input the standardized declaration dataset with the quality flags generated through the "LPIS / GSAA Declaration Dataset Preparation component" (Sen4CAP_DDF-ATBD-LPIS-GSAA_v1.0) to prepare the shapefile that will be used as input data for the grassland mowing product generation through the corresponding component of the "Sen4CAP system orchestrator" (Sen4CAP_DDF_v1.0).

Starting from the standardized declaration dataset (name: **{country} {year}_DeclSTD_quality_indic),** this component generates a new shapefile layer <inputShapeFile> with the following characteristics:

- Type: shape file;
- Naming convention: None.
- Attributes. The shape file contains the following attributes:

[1] NewID: Unique Parcel ID (int)
[2] Ori_hold: Holding ID (string)
[3] Ori_id: Original Parcel ID (string)
[4] Ori_crop: Code of crop type selected as grassland (string)
[5] Area_meter: Parcel size in hectares (float)
[6] mow_n: number of the detected mowings events (int in [0,1,2,3,4])
[7] m1_dstart: date and time of the start mowing (string)
[8] m1_dend: date and time of the end mowing (string)
[9] m1_conf: confidence level (float)
[10] m1_mis: satellite mission (string)
[11] m2_dstart: date and time of the start mowing (string)
[12] m2_dend: date and time of the end mowing (string)
[13] m2_conf: confidence level (float)
[14] m2_mis: satellite mission (string)
[15] m3_dstart: date and time of the start mowing (string)
[16] m3_dend: date and time of the end mowing (string)
[17] m3_conf: confidence level (float)
[18] m3_mis: satellite mission (string)
[19] m4_dstart: date and time of the start mowing (string)
[20] m4_dend: date and time of the end mowing (string)
[21] m4_conf: confidence level (float)
[22] m4_mis: satellite mission (string)
[23] proc: flag for processed parcel (int 0 or 1)
[24] compl (int in [0,1,2])

Note that attributes from [1] to [5] derive from the standardized declaration dataset (name: **{country} {year}_DeclSTD_quality_indic)**, while attributes from [3] to [21] are intialized as zero value (0).

- The LPIS\GSAA shapefile contains the grassland parcels of each country. Following table lists, for each country, the crop codes to be considered for selection of the parcels of interest and the generation of the <inputShapeFile>.

Table 2-3. Grassland crop code and name for each country

Country	Crop code (Oridinal crop ID), name
Czech Republic (CZE)	315, Temporary Grassland 350, Permanent Grassland 3001, Permanent Grassland
Spain (ESP)	2, Alfalfa 85, Grassland pasture
Italy (ITA)	46, Loietto Loglio 51, Lupolina 65, Grassland 79, Vetch 152, Clover 336, Meadow 389, Vetch species 390, Vetch species 460, Grassland with orchid 461, Herbal species 562, Alpha-Alpha 581, Annual grassland 612, Annual grassland 800, Annual grassland 840, Annual grassland 862, Fenugreek 899, Permanent grassland
Lithuania (LTU)	GPŽ, Pasture or meadow, perennial grass up to 5 years DGP, Perennial pastures or meadows 5 years and more GPA, Pasture or meadow, perennial grass up to 5 years, renewed in the current year EPT, Extensive meadows grazing with livestock SPT, Specific meadows 5PT-2, Extensive management of wetlands (direct payments are paid MNP, Aquatic warbler habitats storage in raw and semi-natural grasslands MNS. Aquatic warbler habitats storage in wetlands
Netherlands (NLD)	265, Grassland, permanent 266, Grassland temporarily 331, Grassland, natural. Main function of agriculture 332, Grassland natural. Main function nature. 333, Edge adjacent to permanent pasture or permanent cultivation consisting mainly of permanent grass. 334, Edge adjacent to land mainly consisting of permanent grass. (EA: not managed). 370, Edge adjacent to permanent pasture or permanent cultivation mainly consisting of temporary grass.

Country	Crop code (Oridinal crop ID), name
	372, Edge adjacent to land mainly consisting of temporary grass. (EA: not managed). 383, Grass seed. 1921, Grass sods. 3506, English raaigrass 3509, Festulolium 3512, Italianraaigrass 3513, Westerwoldsraaigrass 3519, Sorghum 3522, Timothee 3523, Veldbeemdgras 3805, Rietzwenkgras industrial grass 3807, Rietzwenkgras other than for industrial grass 3808, Roodzwenkgras
Romania (ROU)	450, Temporary grassland (artificial, sowed on AL < 5 years) 603, Public permanent grasslands used in common 604, Permanent grasslands used in common 605, Public permanent grasslands used individually 606, Permanent grasslands used individually 607, Individually used meadows 608, Public meadows used individually 609, Pasture individual 610, Pasture shared 611, Hay shared mowed 612, Pasture communal but used individually 660, Traditional orchard extensively used by pasturage and or mowing 661, Traditional orchard extensively used as meadow 662, Energy natural meadows 663, Orchard traditional extensive pasture 671, Convert sensitive PP

2.3 SAR data processing

Methods based on SAR coherences have shown potential for change detection in agriculture practicing, and the regularity of S-1 data is well suited for such use. In fact, high coherence values are mainly due to backscattering from the ground and can be linked to mowing and ploughing practices. Similar approaches, but for the detection of mowing events on grasslands, have been recently developed by several authors, where it was demonstrated how coherence increases after grasslands mowing. This is the main principle exploited in the developed technique to detect mowing events. The weekly frequency of the monitoring is achieved by triggering the processing at each new S-1A or S-1B acquisition. Given the last acquisition, a temporal stack of amplitude SAR data and the one formed by coherences are generated and processed. The SAR based detection of the mowing events are combined, through a fusion process, with the detection results from S-2 derived data as described in Section 2.4.1.

2.3.1 SAR feature extraction

This part of the processing receives as input the SAR σ_0 (sigma_nought), the S-1 short-term (6 days) coherences as well as the LPIS/GSAA shape files. The output of this process is the temporal trends of the sigma_nought and coherences averaged over the parcels areas.

The extraction of the temporal trends is performed only for the last six σ_0 and coherences images (VV and VH polarization):

- σ_0 at the times T_0, T_{-1}, T_{-2}, T_{-3}, T_{-4}, T_{-5}
- coherences at the intervals T_0-T_{-1}, T_{-1}-T_{-2}, … T_{-5}-T_{-6}.

These data do not need to have the same grid, in other words, they are not necessarly stacked. The images need only to be geocoded. The temporal series of coherneces and amplitudes is built, image per image, through object based approach. In particular, based on LPIS or GSAA derived segmentation, the average of the calibrated amplitude backscatterer (square root of σ_0) and of the coherences within each parcel is calculated (a morphologic erosion of the parcel segments can be applied). At the end of this process, 4 NxM arrays are generated, where N = number of parcels and M = number of available dates (6 in our case):

- 2 array containing coherences time series (VH and VV) for all parcels;
- 2 array containing σ_0 time series (VH and VV) for all parcels;

This processing is summarized in Figure 2-1.

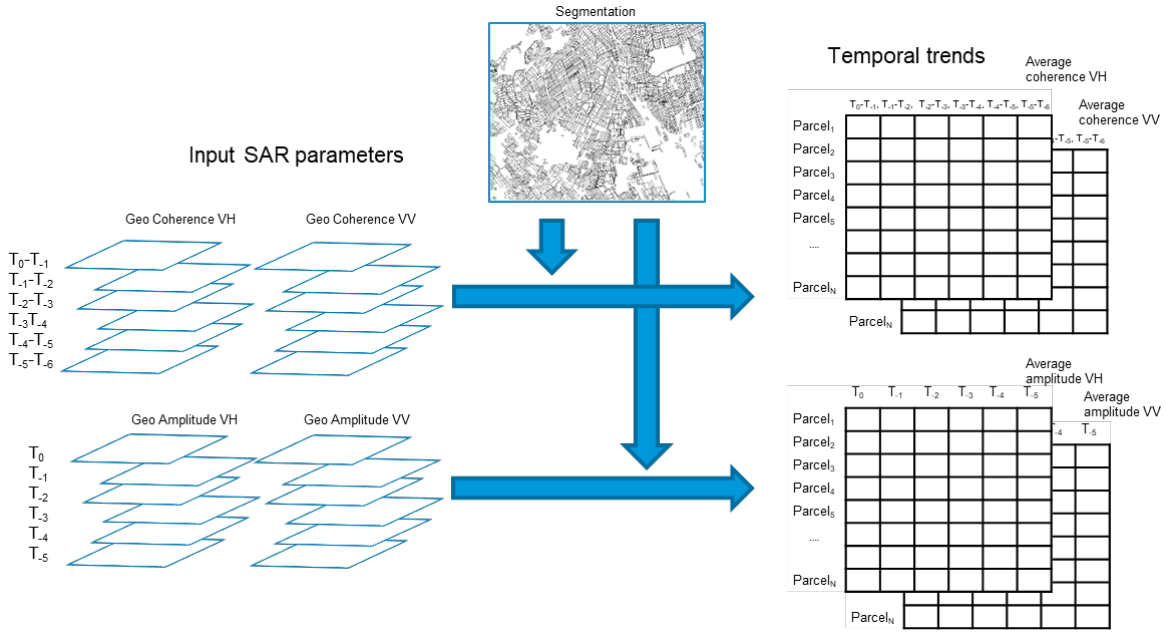

Figure 2-1. Coherence and σ_0 VV and VH time series generation

To maximise the SAR derived information, **both polarization VV and VH** are considered as well as all **ascending and descending** passes. The SAR-based indicators includee backscatter σ_0 temporal profiles and coherence temporal profiles (ascending and descending orbits for dual VV and VH polarization) at 20 m spatial sampling.

It is worth mentioning here the process used to calculate the input coeherence in order to give a definition of the coherence used for the mowing detection. The coherence is calculated in the SAR slant range geometry and at full resolution, through the spatial ensemble performed in a L=8x2 pixels mowing

window. The resulting ground resolution of the coherence is about 8 pixels x 4.9 meters = 39.2 meters in ground range and 2 pixels x 23 meters = 46 meters in azimuth. The coherence is finally resampled and ground projected from the original (radar) sampling and geometry (slant range 2.3 m x az. 14.1 m) to 20 m x 20 m.

The described processing uses the following modules:

- S-1 Data selection: given the last available σ_0 and coherence corresponding respectivelly to the acquisition times T_0 and T_0-T_{-1}, the previous data are identified and searched in the data storage (T_{-1}, T_{-2}, T_{-3}, T_{-4}, T_{-5} acqusition times for σ_0 and T_{-1}-T_{-2}, … T_{-5}-T_{-6} for cohereneces). In this step, the footprint of the data and its geometry/projection is retrieved.
- Parcel rasterization: the input LPIS/GSAA shapefile, filtered over the grassland parcels, is rasterized over the footprint and geometry/projection of the SAR data. This step generates a segmentation raster on which each segment is burned with the FID (Feature Identifier) value of the parcel in the LPIS/GSAA shapefile.
- Temporal trends generation: the temporal trends are calculated for all parcels and for each time as the spatial average over the parcel as identified by the corresponding segment in the segmentation. The four temporal trends of the avearge SAR indexes are stored in four array structures (one for index) with size NxM, where N = number of parcels and M = number of available dates (6 in our case).

2.3.2 SAR mowing detection

The mowing detection based on SAR data is based on the following principles:

- high grass makes low the measured coherence, because incoherent.
- coherence across a mowing are low because the scene between pre and post mowing is is changed.
- coherence based on both images after the mowing is expected higher because corresponding to two scenes with low level of the grass and therefore with a component of the backscattere coming from the soil which should be more coherent of the grass leafs.

On the base of these principles, the main mechanism used to detect a mowing is to identify sudden increasing of the coherences through change detection in the coherences temporal trends.

These assumptions have been validated by statistical analysis performed with S-1 data[1]. However, there are different aspects that makes the detectability of a grassland mowing event in some cases difficult. For instance, rain on the scene and different moisture level between acqsuitions makes coherences remaining at low values after a mowing event or, on the contrary, the coherence suddenly increases because one of such conditions ceases and a mowing is detected erroneously.

The change detection is performed independently for each temporal trend of cohereneces of each parcel.

Therefore, for each parcel, the time series are processed to detect a strong increase of the last coherence (T_0-T_{-1}- VH and VV) with respect to the previous coherences (T_{-1}-T_{-2}, …, T_{-5}-T_{-6}). To this aim, a smooth trend of the previous 5 coherences is calculated by applying a linear fit of these 5 values. This trend is used to predict the value at the time T_{-1}-T_{-2}, which will be used to assess the change with the last coherence T_0-T_{-1}. The standard deviation (σ) of the residual fitting errors gives information about the

[1] Tamm, T., Zalite, K., Voormansik, K., & Talgre, L. (2016). Relating Sentinel-1 interferometric coherence to mowing events on grasslands. Remote Sensing, 8(10), 802.

variability of the coherence trend around the fitted line and is used to calculate the threshold. The detection is finally calculated by applying the following thresholding criterion:

$Cohe(t=T_0\text{-}T_{-1}) > Cohe\_fit(t=T_{-1}\text{-}T_{-2}) + k\,\sigma$, where:

- $Cohe(t=T_0\text{-}T_{-1})$ is the last coherence,
- $Cohe\_fit(t=T_{-1}\text{-}T_{-2})$ is the value of the fitted coherence at the time $T_{-1}\text{-}T_{-2}$
- k(PFA) is calculated to have a constant probability of false alarm PFA (typically below 1x10e-4).

This CFAR criterion assumes that the residual fitting errors are normally distributed.

The processing depends on the following steps:

- Local trend fit: a linear fit of the the previous coherences ($T_{-1}\text{-}T_{-2}$, …, $T_{-5}\text{-}T_{-6}$) is performed in order to assess i) prediction at the time $T_{-1}\text{-}T_{-2}$, $Cohe\_fit(t=T_{-1}\text{-}T_{-2})$ and ii) standard deviation (σ) of the residual fitting errors.
- CFAR detection: detection of coherence increasing by using as adaptive threshold $th = k\,\sigma$:
  $Cohe(t=T_0\text{-}T_{-1}) > Cohe\_fit(t=T_{-1}\text{-}T_{-2}) + k\,\sigma$
- Confidence assessment: the difference $Cohe(t=T_0\text{-}T_{-1})$ - $Cohe\_fit(t=T_{-1}\text{-}T_{-2})$ is related to the confidence level of the detection.

The detections and the related confidences are stored in two array structures with size NxM, where N = number of parcels and M = number of available dates (6 in our case).

2.3.3 Pseudo-Code

2.3.3.1 Library Import

 # standard library

```
import os, glob
import sys
import re
import configparser
import numpy as np
import dateutil.parser
import pandas as pd
import scipy
import time
import dateutil.parser
import gdal
from osgeo import osr, ogr
gdal.UseExceptions()
ogr.UseExceptions()
osr.UseExceptions()
```

 # new routines

```
import S1_gmd
import fusion
```

2.3.3.2 Input parameters

 # processing parameters

```
options_layer_burning=['ALL_TOUCHED=False'] # option for gdal_rasterize
pfa_fit = 3.0e-7# constant probability of false alarm
fit_smpl_nt = 5 # number of data on which calculate the linear fitting
min_cohe_var = 0.024  # theoretically expected variance of the coherence
no_mowing_after_det = 60 # No mowing are expected before than 60 days
validity_temporal_range_str = ('20190401 00:00:00', '20191031 23:59:59') # temporal range used for mowing detection
cohe_ENL = 20*5 # number of SAR Equivalent Number of Look
erode_pixels = 0 # number of pixel buffer to remove in morphological erosion
```

2.3.3.3 Data, Orbits and paths (to be extracted from last preprocessed data)

 # Input preprocessed SAR data parameters

```
orbit_list = 015, 088, 161, 059, 139, 037, 110, 008 # example of S1 orbit list for Netherlands
orbit_type_list = ASC, ASC, ASC, ASC, DESC, DESC, DESC, DESC # example of list of S1 orbit types for Netherlands
pol_types = VH, VV # list of S1 orbit polarization
data_types = AMP, COHE # list of S1 data types
invalid_data = nan
```

 # Files and paths

```
sarDataRoot = <root directory where input preprocessed SAR data are stored>
outputDir = < output directory where the output shape file is stored>
outputShapeFile = <name of the output shapefile>
segmentsFile = <name of the input shapefile used for segmentation>
```

 # Attribute names

```
seg_parcel_id_attribute = 'parcel_id' # name of the attribute for extracting parcel id
```

2.3.3.4 File list generation and parameter extraction

 # The list of the SAR data file names, coherences and amplitudes sorted for acqsuition date and time, is extracted from the sarDataRoot directory

```
    # setting file name parsing tools
    keys = ['file_name', 'satellite', 'master_date', 'master_time', 'slave_date', 'slave_time', 'pol', 'orbit', 'data_type']

    get_par_from_file = re.compile('(SEN4CAP_L2A_PRD_(S[0-9]{1,2})_[0-9]{8}T[0-9]{6}_V([0-9]{8})T([0-9]{6})_([0-9]{8})T([0-9]{6})_([VH]{2})_([0-9]{3})_([A-Z]{3,4})'+file_ext+')')

    # file list generation from data dir
    file_list = glob.glob(os.path.join(os.path.join(sarDataRoot,'*'),'*'+file_ext))
```

```
    # extract data file names and dates from file list and for specific orbits,
polarization and data type
    par_list = S1_gmd_v2.read_file_list(file_list, get_par_from_file, keys,
[orbit_list,], polType, dataType)

    # verify if there are data for specific orbits
    if len(par_list) == 0:
        print("There are NO data for the specific orbit", orbit_list)
        return 2

    # put filename and data parameters in a pandas df
    df = pd.DataFrame.from_records(par_list, columns=keys).drop_duplicates(keys[0])

    # add the orbit type column
    orbit_type_dict = {orbit: orbit_t for orbit, orbit_t in zip([orbit_list,],
[orbit_type,])}
    df['orbit_type'] = df.apply(lambda x: orbit_type_dict[x['orbit']], axis=1)
```

2.3.3.5 Extraction of dates and times

 # Dates and times are extracted from file names, converted in date-time formats and stored in a list structure

```
    # date and time conversion from str
    df['master_date_time'] = pd.to_datetime(df.apply(lambda x:
x['master_date']+'T'+x['master_time'], axis=1), yearfirst=True, dayfirst=False)
    df['slave_date_time'] = pd.to_datetime(df.apply(lambda x:
x['slave_date']+'T'+x['slave_time'], axis=1), yearfirst=True, dayfirst=False)
    df.drop(['master_date', 'master_time', 'slave_date', 'slave_time'], axis=1,
inplace=True)

    # add acq_time1 acq_time_old columns
    df['old_acq_time'] = df.apply(lambda x: min(x['master_date_time'],
x['slave_date_time']), axis=1)
    df['acq_time'] = df.apply(lambda x: max(x['master_date_time'],
x['slave_date_time']), axis=1)
```

2.3.3.6 Data selection based on dates

 # The files out of the temporal range of interest are discarded

```
    # retrieval of the date/time range
    if older_acq_date:
        validity_temporal_range_str = [older_acq_date+"T000000",
new_acq_date+"T235959"]
    else:
        validity_temporal_range_str = [new_acq_date+"T000000",
new_acq_date+"T235959"]

    # conversion from string to datetime type
    validity_temporal_range_str = [dateutil.parser.parse(date_str, yearfirst=True,
dayfirst=False) for date_str in validity_temporal_range_str]
    select_date_interval = [validity_temporal_range_str[0] -
datetime.timedelta(days=S1_time_interval*stat_smpl_n),
validity_temporal_range_str[1]]

    # date selection in the validity date/time range
    valid_date_mask = (df['acq_time'] >= select_date_interval[0]) & (df['acq_time']
<= select_date_interval[1])
    df = df.loc[valid_date_mask]
```

```
    # generate lists of files and dates
    data_list = df['file_name'].values
```

2.3.3.7 Removal of corrupted files

```
    # remove corrupted files
    data_list = S1_gmd_v2.remove_corrupted_files(list(data_list))
```

2.3.3.8 Generation of segmentation map from input shapefiles and extent extraction

 # Extraction of projection of the input shapefile

```
    # Get projection from shape file
    ogr_data = ogr.Open(segmentsFile)
    Layer = ogr_data.GetLayer(0)
    spatialRef = Layer.GetSpatialRef()
    dst_srs = spatialRef.ExportToWkt()

    # Get shape file extent
    (min_x, max_x, min_y, max_y) = Layer.GetExtent()
    print("shape file extent", (min_x, min_y, max_x, max_y))

    # Some gdalwarp parameters
    resampling = gdal.GRA_Bilinear
    error_threshold = 0.125  # error threshold --> use same value as in gdalwarp
```

2.3.3.9 Generation of the GDAL Virtual Raster of the SAR data on the shapefile projection

```
    # make vrt of all data without considering extension of the shape file
    print('Make virtual raster of input data (without considering extension of the
shape file)')
    output_vrt_tmp = os.path.join(output_tmp_dir, "data_cube_tmp.vrt")
    images_n = S1_gmd_v2.make_vrt(data_list, dst_srs, output_tmp_dir,
output_vrt_tmp, outputBounds=None,
                                  srcNodata=invalid_data, resampling=resampling,
error_threshold=error_threshold)

    if images_n > 0:
        vrt_data = gdal.Open(output_vrt_tmp)
    else:
        print('Empty VRT. Does orbit intersect segment layer?')
        return 1
```

2.3.3.10 Extraction of the gdal virtual raster extent

```
    # Get virtual raster extension
    geoTr = vrt_data.GetGeoTransform()
    r_min_x = geoTr[0]
    r_max_y = geoTr[3]
    r_max_x = r_min_x + geoTr[1]*vrt_data.RasterXSize
    r_min_y = r_max_y + geoTr[5]*vrt_data.RasterYSize
    r_extent = (r_min_x, r_min_y, r_max_x, r_max_y)
    del vrt_data

    print("virtual raster extent", (r_min_x, r_min_y, r_max_x, r_max_y))
```

2.3.3.11 Extraction of the intersection between gdal virtual raster extent and shapefile extent

```
    # Calculate extent intersection between virtual raster and shape file
    outputBounds = (np.maximum(min_x, r_min_x),
                    np.maximum(min_y, r_min_y),
                    np.minimum(max_x, r_max_x),
                    np.minimum(max_y, r_max_y))
```

2.3.3.12 Generation of the GDAL Virtual Raster of the SAR data on the shapefile projection and on the intersection extent

 # Finally, a new gdal VRT is generated with the extent equal to the intersection between input shape file and SAR data available

```
    # make vrt over the intersection extent
    print('Make virtual raster of input data (considering intersection between
extensions of i- the shape file, ii- the virtual raster)')
    output_vrt = os.path.join(output_tmp_dir, "data_cube.vrt")
    images_n = S1_gmd_v2.make_vrt(data_list, dst_srs, output_tmp_dir, output_vrt,
outputBounds=outputBounds,
                                  srcNodata=invalid_data, resampling=resampling,
error_threshold=error_threshold)
    if images_n > 0:
        vrt_data = gdal.Open(output_vrt)
    else:
        print('Empty VRT. Does orbit intersect segment layer?')
        return 1
```

2.3.3.13 Generation of segmentation map from input shapefiles

 # Input segmentation shapefile is projected on the gdal virtual raster of the SAR data

 # Output segmentation raster file name building

```
segmentsOutputDir=os.path.join(outputDir, "segments")

segmentsOutRaster                    =                    os.path.join(segmentsOutputDir,
os.path.basename(segmentsFile)[:-4]+'_raster')
```

 # make directories

```
try:

    os.mkdir(segmentsOutputDir)

except(OSError):

    print('directory ',segmentsOutputDir,' exists...moving on')
```

 # Generate segmentation raster map from input shapefile. The segmentation shapefile is rasterized over the raster coheVV_list[0]

```
burned_pixels,  seg_attributes  =  S1_gmd_v2.layer2mask(segmentsFile,  output_vrt,
segmentsOutRaster, layer_type='segments', options=options_layer_burning)
```

2.3.3.14 Load Data

 # load segments

```
    # load segments
    gdal_data =  gdal.Open(segmentsOutRaster)
    segments = gdal_data.ReadAsArray()
```

```
    segments_geo_transform = gdal_data.GetGeoTransform()
    segments_projection = gdal_data.GetProjection()
```

2.3.3.15 Morphological erosion of segments

 # Morphological erosion is applied to the segmentation mask if erode_pixels > 0

```
    # erode segments

    eroded_segs = np.copy(segments)
    if erode_pixels > 0:
        print('Apply erosion')
        # first step: separate segments
        cross = np.array([[0,1,0],[1,1,1],[0,1,0]])
        min_seg = scipy.ndimage.minimum_filter(segments,footprint=cross)
        max_seg = scipy.ndimage.maximum_filter(segments,footprint=cross)
        segs_borders = (max_seg-min_seg)>0
        eroded_segs[segs_borders] = 0

        # second setp: erode
        if erode_pixels > 1:
            erosion_mask = scipy.ndimage.morphology.binary_erosion(eroded_segs>0,
structure=cross,iterations=(erode_pixels-1))
            eroded_segs = eroded_segs*erosion_mask
```

2.3.3.16 Extraction of segment id and their parameters

 # Extraction of the segments

```
    # segments
    unique_segments = np.unique(eroded_segs)
    if unique_segments[0] == 0: # 0 label corresponds to not valid segments and it
will be removed
        unique_segments = unique_segments[1:]
    print(unique_segments.shape)
    seg_pixels_num = np.array(ndimage.sum(eroded_segs>0, eroded_segs,
index=unique_segments))
```

2.3.3.17 Extraction data parameters from gdal virtual raster

```
    # extract data parameters and dates from file_list in the vrt data
    print('Extract data parameters and dates from file_list in the vrt data')
    par_list = S1_gmd_v2.read_file_list(vrt_data.GetFileList()[1:],
get_par_from_file, keys, [orbit_list,], polType, dataType)

    print('Put data parameters in a pandas structure')

    # put data parameters in a pandas df
    print("fill pandas structure")
    vrt_df = pd.DataFrame.from_records(par_list, columns=keys)

    # add the orbit type column
    orbit_type_dict = {orbit: orbit_type for orbit, orbit_type in
zip([orbit_list,], [orbit_type,])}
    vrt_df['orbit_type'] = vrt_df.apply(lambda x: orbit_type_dict[x['orbit']],
axis=1)

    # date and time conversion from str
    vrt_df['master_date_time'] = pd.to_datetime(vrt_df.apply(lambda x:
x['master_date']+'T'+x['master_time'], axis=1), yearfirst=True, dayfirst=False)
    vrt_df['slave_date_time'] = pd.to_datetime(vrt_df.apply(lambda x:
x['slave_date']+'T'+x['slave_time'], axis=1), yearfirst=True, dayfirst=False)
```

```
    vrt_df.drop(['master_date', 'master_time', 'slave_date', 'slave_time'], axis=1,
inplace=True)

    # add acq_time1 old_acq_time
    vrt_df['old_acq_time'] = vrt_df.apply(lambda x: min(x['master_date_time'],
x['slave_date_time']), axis=1)
    vrt_df['acq_time'] = vrt_df.apply(lambda x: max(x['master_date_time'],
x['slave_date_time']), axis=1)
    vrt_df['old_acq_date'] = vrt_df.apply(lambda x: x['old_acq_time'].date(),
axis=1)
    vrt_df['acq_date'] = vrt_df.apply(lambda x: x['acq_time'].date(), axis=1)
```

2.3.3.18 Feature Extraction based on segmentation

```
    #selection of the average method
    stat_p = scipy.ndimage.mean

    # Write list of dataframe extracting, for each image in the vrt, the averages
and the counts of amplitudes and coherences within the segments
    list_df = S1_gmd_v2.load_stats(vrt_data, vrt_df, eroded_segs, unique_segments,
seg_attributes, seg_parcel_id_attribute, stat_p, invalid_data)

    # Concatenate all pandas elements in the list
    data_df = pd.concat(list_df, ignore_index=True)

    # Sort wrt count
    data_df.sort_values(['count'], ascending=[False], inplace=True)

    # Aggregate with respect the same date and same parcel_id by keeping the stats
associated with the highest count
    data_df = data_df.groupby(['acq_date', 'data_type',
seg_parcel_id_attribute]).aggregate('first')

    # Rearrange structure to put dates and data_types as columns
    data_df = pd.pivot_table(data_df, values=['mean','count'],
index=[seg_parcel_id_attribute],
                             columns=['orbit', 'pol', 'data_type', 'acq_date'],
dropna=False)

    # Sort with respect the dates (from early to late)
    data_df = data_df.sort_values(by=['acq_date'], axis=1, ascending=True)
```

2.3.3.19 Extraction of the temporal series

```
    typical_acq_time = {'asc': locAcqTimeASC, 'desc': locAcqTimeDESC, 'ASC':
locAcqTimeASC, 'DESC': locAcqTimeDESC}
    orbit_type_dict = {orbit: orbit_type for orbit, orbit_type in
zip([orbit_list,],  [orbit_type,])}

    parcel = data_df.index.values

    # generate array with temporal series
    ampVV_seg = {}
    ampVH_seg = {}
    coheVV_seg = {}
    coheVH_seg = {}

    orbit_list = {x[1] for x in data_df.columns}
    for o in orbit_list:
        if ('VV' in polType) and ('AMP' in dataType):
            ampVV_seg[o] = data_df.xs(key=('mean', 'VV', 'AMP', o), level=(0,
'pol', 'data_type', 'orbit'), axis=1).values
```

```
            dates_str = data_df.xs(key=('mean', 'VV', 'AMP', o), level=(0, 'pol',
'data_type', 'orbit'), axis=1).columns.values
            ampVVDateList =
[datetime.datetime.combine(pd.Timestamp(d).to_pydatetime(),
datetime.time(hour=int(typical_acq_time[orbit_type_dict[o]][:2]),
minute=int(typical_acq_time[orbit_type_dict[o]][3:5]))) for d in dates_str]

        if ('VH' in polType) and ('AMP' in dataType):
            ampVH_seg[o] = data_df.xs(key=('mean', 'VH', 'AMP', o), level=(0,
'pol', 'data_type', 'orbit'), axis=1).values
            dates_str = data_df.xs(key=('mean', 'VH', 'AMP', o), level=(0, 'pol',
'data_type', 'orbit'), axis=1).columns.values
            ampVHDateList =
[datetime.datetime.combine(pd.Timestamp(d).to_pydatetime(),
datetime.time(hour=int(typical_acq_time[orbit_type_dict[o]][:2]),
minute=int(typical_acq_time[orbit_type_dict[o]][3:5]))) for d in dates_str]

        if ('VV' in polType) and ('COHE' in dataType):
            coheVV_seg[o] = data_df.xs(key=('mean', 'VV', 'COHE', o), level=(0,
'pol', 'data_type', 'orbit'), axis=1).values
            dates_str = data_df.xs(key=('mean', 'VV', 'COHE', o), level=(0, 'pol',
'data_type', 'orbit'), axis=1).columns.values
            coheVVDateList2 =
[datetime.datetime.combine(pd.Timestamp(d).to_pydatetime(),
datetime.time(hour=int(typical_acq_time[orbit_type_dict[o]][:2]),
minute=int(typical_acq_time[orbit_type_dict[o]][3:5]))) for d in dates_str]
            coheVVDateList1 = [d - datetime.timedelta(days=S1_time_interval) for d
in coheVVDateList2]

        if ('VH' in polType) and ('COHE' in dataType):
            coheVH_seg[o] = data_df.xs(key=('mean', 'VH', 'COHE', o), level=(0,
'pol', 'data_type', 'orbit'), axis=1).values
            dates_str = data_df.xs(key=('mean', 'VH', 'COHE', o), level=(0, 'pol',
'data_type', 'orbit'), axis=1).columns.values
            coheVHDateList2 =
[datetime.datetime.combine(pd.Timestamp(d).to_pydatetime(),
datetime.time(hour=int(typical_acq_time[orbit_type_dict[o]][:2]),
minute=int(typical_acq_time[orbit_type_dict[o]][3:5]))) for d in dates_str]
            coheVHDateList1 = [d - datetime.timedelta(days=S1_time_interval) for d
in coheVHDateList2]

    if ('VV' in polType) and ('AMP' in dataType):
        ampVV_seg = ampVV_seg[o]
    if ('VH' in polType) and ('AMP' in dataType):
        ampVH_seg = ampVH_seg[o]
    if ('VV' in polType) and ('COHE' in dataType):
        coheVV_seg = coheVV_seg[o]
    if ('VH' in polType) and ('COHE' in dataType):
        coheVH_seg = coheVH_seg[o]
```

2.3.3.20 Make dictionary of the segments and its inversion

```
    # make dictionary segment --> seg_parcel_id_attribute
    seg_attributes = {i: {seg_parcel_id_attribute: p} for i, p in
enumerate(parcel)}

    # inverti dizionari
    inv_seg_dct = {v[seg_parcel_id_attribute]:k for k,v in seg_attributes.items()}
```

2.3.3.21 Detection: Constant false alarm rate (CFAR) initialization

 # Calculation of the k_fact depending on the input probability of false alarm parameter pfa

```
k_fact = np.sqrt(2)*scipy.special.erfinv(1. - 2.*pfa)
```

2.3.3.22 Detection: fitting and detection on VV temporal trends of the coherences

 # The input temporal trends, except the last data, are linearly fitted

```
data_seg_pred, data_seg_std = S1_gmd.temporal_linear_fit(coheVV_seg, coheDateList1,
stat_smpl_n, linear_fit=True) # data_seg_pred is the last value of the fitted line
(at the time T-1-T-2), predicted, data_seg_std is the associated error standard
deviation
```

 # Calculate detection and confidences. The CFAR thresholding criterium is applied: coheVV_seg((t=T0-T-1) > data_seg_pred(t=T-1-T-2) + k_fact data_seg_std,

```
coheVV_det_cube_fit = S1_gmd.CFAR_detection(coheVV_seg, k_fact, data_seg_pred,
data_seg_std, saturate_sigma_seg=saturate_sigma) # coheVV_det_cube_fit is an array
with shape NxM, where N=number of segments, M=number of times. In the last column
(last time), it contains 0 for the segments in which no mowing are detected or contains
the confidence levels (0<conf<1) if a mowing event is detected.
```

2.3.3.23 Detection: fitting and detection on VH temporal trends of the coherences

 # The input temporal trends, except the last data, are linearly fitted

```
data_seg_pred, data_seg_std = S1_gmd.temporal_linear_fit(coheVH_seg, coheDateList1,
stat_smpl_n, linear_fit=True) # data_seg_pred is the last value of the fitted line
(at the time T-1-T-2), predicted, data_seg_std is the associated error standard
deviation
```

 # Calculate detection and confidences. The CFAR thresholding criterium is applied: coheVH_seg((t=T0-T-1) > data_seg_pred(t=T-1-T-2) + k_fact data_seg_std,

```
coheVH_det_cube_fit = S1_gmd.CFAR_detection(coheVH_seg, k_fact, data_seg_pred,
data_seg_std, saturate_sigma_seg=saturate_sigma) # coheVV_det_cube_fit is an array
with shape NxM, where N=number of segments, M=number of times. In the last column
(last time), it contains 0 for the segments in which no mowing are detected or contains
the confidence levels (0<conf<1) if a mowing event is detected
```

 # detection on coherences have 1-acquisition delay. Remove this delay

```
coheVV_det_cube_fit = np.roll(coheVV_det_cube_fit, -1, axis=1)

coheVV_det_cube_fit[:,-1] = 0
```

2.3.3.24 Normalization of the confidence index for VV and VH detections

 # Calculate detection reliability index as normalized index. For SAR data, the confidence index is bounded within (0.0, 0.5)

```
coheVH_det_cube_fit[coheVH_det_cube_fit>0]                                    =
S1_gmd.norm_fun(coheVH_det_cube_fit[coheVH_det_cube_fit>0],   alpha,   bounds=(0.0,
0.5))

coheVV_det_cube_fit[coheVV_det_cube_fit>0]                                    =
S1_gmd.norm_fun(coheVV_det_cube_fit[coheVV_det_cube_fit>0],   alpha,   bounds=(0.0,
0.5))
```

2.3.3.25 Calculate fused confidences exploiting VV and VH detections

 # Detection on VV coherences are used to update the confidence indexes of the mowing detections, which are based on VH coherences

```
det_cub = np.copy(coheVH_det_cube_fit) # make a copy of the VH detection data structure
det_cube[det_cube>0] = np.maximum(det_cube[det_cube>0], coheVV_det_cube_fit[det_cube>0])
```

2.3.3.26 Write detection shape file results

 # If outputShapeFile does not exist, it is assumed that the input segmentsFile can be used to derive geometry and it is cloned on the output directory

```
if not os.path.exists(outputShapeFile):
    fusion.cloneAndUpdateShapefile(segmentsFile, outputShapeFile)
```

 # The new detections are added in the output shape file outputShapeFile

```
fusion.writeDetections_S1(outputShapeFile, unique_segments, det_cube, coheDateList1, coheDateList2, mission_id='S1', max_dates=4, minimum_interval_days=30)
```

2.3.3.27 Calculate compliancy

 # Run compliancy calculation (for details see section 2.5)

```
        fusion.do_compliancy(outputShapeFile, cnt_crop_code, cnt_crop_TR, cnt_crop_rule)
```

2.4 Optical data processing

S-2 data series are used to calculate trends of indices related to the biomass (for example NDVI) and to detect the sudden decreasing of these trends to identify a mowing event.

To this regard, it is worth noting that, depending on the European regional area and in particular in the Mediterranean areas, decreasing of NDVI could occur due to grass drying out before and during Summer, but it has a slower dynamic with respect to mowing events. To deal with these phenomena the developed algorithm exploits an adaptive model of the phenology of the grassland, which is dependent on the geographical area. The mowing detection is therefore based on the identification of decreasing of VIs with respect to the expected model of unmowed grassland for that area. The detection results are fused with SAR results as described in the Section 2.4.1.

2.4.1 Vegetation Indexes feature extraction

Any S-2 NDVI image available in the shapefile footprint at the time T0 are made available, as well as the associated LAI and FCover VIs. The cloud and shadow masks have been already generated and applied to the available VIs images. Indeed, the VIs images have invalid values in areas covered by cloud or cloud shadow. The input images span a temporal period ending with the last indexes (at the time T0) and including all previous ones (at the times T_{-1}, T_{-2}, T_{-3}, …, $T_{\text{first acquisition}}$) from a “first acquisition” date, where the date of the “first acquisition” depends on the monitoring period defined for grassland mowing detection. For instance, if the monitoring period is between April and October, the first April is selected as the first useful acqsuiston date. The VIs data need to be geocoded in order to allow the extraction of the temporal series of the VI indexes.

As for the SAR based indexs, the geocoded VIs are processed to generate temporal trends of VIs averaged over parcel segments for all parcels. To this aim, LPIS/GSAA layer is used to identify the parcells and to derive segmentation raster with the same grid of the VIs data.

At the end of this process, one NxM array is generated for each VI, where N = number of parcels and M = number of available dates from the start of the monitoring period

This processing is summarized in Figure 2-2.

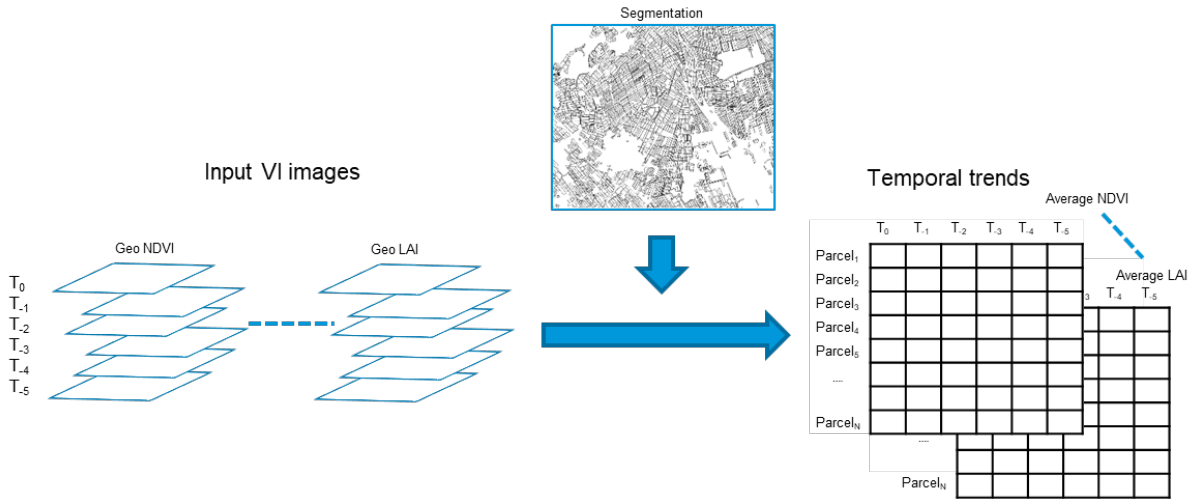

Figure 2-2. VIs time series generation

The described processing uses the following modules:

- S-2 Data selection: all acquisition times available from the "first acquisition" to the last one, T_0, are identified and searched in the data storage. In this step, the footprint of the data and its geometry/projection is retrieved.
- Parcel rasterization: the input LPIS/GSAA shapefile, filtered over the grassland parcels, is rasterized over the footprint and geometry/projection of the Vis data. This step generates a segmentation raster on which each segment is burned with the FID (Feature Identifier) value of the parcel in the LPIS/GSAA shapefile.
- Temporal trends generation: the temporal trends are calculated for all parcels and for each time as the spatial average of the VIs over the parcel as identified by the corresponding segment in the segmentation. The temporal trends of the avearge VIs are stored in array structures (one for index) with size NxM, where N = number of parcels and M = number of available dates.

2.4.2 Vegetation Indexes mowing detection

For each parcel, the time series are processed to detect a decrease in the VI value (T_0) with respect to the last sensed VI or respect to a pre-calculated model. Indeed, two approaches are considered to build a referement with respect to which the VI decreasing is assessed: i) the use of the last cloud free sensed VI or ii) the use of a pre-calculated model. Bothy approaches have advantages and disadvantages here quickly described:

- The use of the last cloud free sensed VI (without pre-calculated model):
  - PROS: this approach is simple and very effective in Mediterranean areas where cloud free acqsuisitios are frequent and even when the grassland phenology is very complex (more rise and decreasing phases depending on whether conditions)

  - CONS: its application on cloudy northen regions prevents the effective detection on the few cases in which the surface is observed.
- The use of a pre-calculated model:
  - PROS: it is effective in cloudy northen regions in which the temporal series is not very dense due to frequent cloud coverage and where the grassland phenology has a standard behaviour (one rising phase and one decreasing phase respectivelly at the start and at the end of the phenology season)
  - CONTRA: in Mediterranean regions, where the grassland phenology can be complex (more rise and decreasing phases) the model cannot effectively calculated and it is too much depending on actual whether conditions.

The grassland mowing detection can be performed using one of the two approaches depending on the geographical regions. The default mode is based on the use of the last cloud free sensed VI (without pre-calculated model).

2.4.2.1 Model calculation

The unmowed grassland model is calculated and used only for the regions which have frequent cloud coverage and in which the grassland phenology has a standard behaviour (one rising phase and one decreasing phase respectivelly at the start and at the end of the phenology season). Indeed, in cloudy regions, only few data per year are generally usable, this is the reason why a model characterizing the VI trend of unmowed grassland is necessary in order to have an reasonable level of detectability of the mowing events. To this aim, if enabled, a model of the unmowed grassland is built based on the assumption that in large areas and in temporal interval of few days (less than 1 week), the mowing is never performed simultaneously in more than 90% parcels in the territory.

The model generation depends on the following modules:

- VI model estimation: This module is run only at the first execution and requires the availability of VIs for one or more previous years.

  The VI model estimation takes into consideration regional criterium and VI temporal trends of the previous years. It takes three sequential steps.

  Step 1: For each S2 granule (regional criterium), over all grassland parcels and at each available time t of the previous year, the following parameters are evaluated:

  - n_t: number of samples VIs at the time t;
  - $p_{95,t}$: 95th percentile of the VIs at the time t;
  - m_t: mean of the VIs the time t;
  - σ_t: standard deviation of the VIs the time t.

  If more than one year of VIs are available in the past, the extraction of the parameters is independently performed for each year. Therefore, for the same S2 granule, a set of temporal series of $\{n_t\}_{t \in Yi}$, $\{p_{95,t}\}_{t \in Yi}$,$\{m_t\}_{t \in Yi}$ and $\{\sigma_t\}_{t \in Yi}$: is retrieved for each past year Y_i , i $\in$ {-1, -2, …}.

  Step 2: The temporal series of $\{n_t\}_{t \in Yi}$, $\{p_{95,t}\}_{t \in Yi}$,$\{m_t\}_{t \in Yi}$ and $\{\sigma_t\}_{t \in Yi}$ retrieved for each past year are merged in one set of temporal series of $\{n_t\}_{t \in Y}$, $\{p_{95,t}\}_{t \in Y}$,$\{m_t\}_{t \in Y}$ and $\{\sigma_t\}_{t \in Y}$, where $Y=\cup_i Y_i$, representing the temporal behaviour characterizing statistically one year. The temporal series will contain all t's from all considered past years without repetitions. The repetitions are avoided, by retrieving a unique set of of n_{ti}, $p_{95,t}$, m_t and σ_t from the same times in different years (e.g. n_{t1}, $p_{95,t1}$, m_{t2}, n_{t2}, $p_{95,t2}$, m_{t2} , …) using the following formulas:

  - $n_t = n_{t1} + n_{t2} + n_{t1} + \ldots$;

- $p_{95,t}$: $\sum(n_{ti}/n_t)\ p_{95,ti}$;
- m_t: $\sum(n_{ti}/n_t)\ m_{ti}$;
- σ_t: *sqrt*(*pooled_variance*(n_{t1}, $p_{95,t1}$, m_{t2}, n_{t2}, $p_{95,t2}$, m_{t2} , …)).

The number of samples $\{n_t\}_{t \in Y}$ at each time is used to assess the reliability of the corresponding estimated 95th percentiles. Therefore, times t's having a small number of samples are discarded.

The remaining statistically significative times t's give the final temporal series of $\{p_{95,t}\}_{t \in \underline{Y}}$, and $\{\sigma_t\}_{t \in \underline{Y}}$, where $\underline{Y} \subset Y$ is the collection of statistically significative times t's.

Step 3: The temporal series of $\{p_{95,t}\}_{t \in \underline{Y}}$ is finally used to estimate the parameters of the double-logistic function fitting such series. The double-logistic function (https://explorer.earthengine.google.com/#detail/LANDSAT%2FLC8_L1T_8DAY_NDVI) has the following form and is characterized by 6 parameters (VI_W, VI_M, m_s, m_a, t_S, t_A):

$$VI(t) = VI_W + (VI_M - VI_W)\left(\frac{1}{1+\exp[-m_S(t-t_S)]} + \frac{1}{1+\exp[m_A(t-t_A)]} - 1\right) \quad (2.1)$$

where $VI(t)$ is the retrieved $p_{95,t}$, VI_W is the winter (minimum) value, VI_M is the maximum VI value, t_S is the increasing inflection point (also referred to as spring date), t_A is the decreasing inflection point (also referred to as autumn date), m_S is the rate of increase at t_S inflection point and m_A is the rate of decrease at the descending inflection point. The application of the double-logistic model relies on the assumption that the grassland phenology is simply given by one rising phase and one decreasing phase respectivelly at the start and at the end of the phenology season. If this assumption fails, the resulting model cannot be representative of the actual grass phenology.

The resulting double-logistic function is the model used to perform the detections togheter the standard deviations $\{\sigma_t\}_{t \in \underline{Y}}$ which are used to assess the reliability of the model at each time.

- Model adjustment: At each new detected VI(t) the model is updated, by rescaling it, following these rules:
  - For each parcel, only at the first valid VI (first cloud free parcel VI), the model is rescaled, by applying a rescaling factor f, such that at the first valid VI the rescaled model passes exactly through the VI value: model=model(t; p, f)
  - If VI(t) >model(t; p, f_{t-1}): the model is rescaled, with a new f=f_t, in order to force VI(t)=model(t; p, f_t).
  - If a grassland mowing has been detected at the time t-1 (VI(t-1)< model(t-1; p, f_{t-1})-kσ_{t-1}, see below detection module, the model is rescaled with a new f=f_t in order to force VI(t)=model(t; p, f_t).

2.4.2.2 Grassland mowing detection

The detection step of the algorithm depends slightly if the pre-calculated model is used or not.

If the model is used, the mowing is detected applying a threshold on the decreasing of the current VI with respect to the precalculated model of the unmowed grassland. These are the steps of the algorithm:

- Detection: A new grassland mowing is deteceted applying this thresholding criterion:

- VI(t) < model(t; p, f_{t-1}) – k σ_t, where:
  - VI(t) is the last VI value
  - model(t; p, f_{t-1}) is the value of the VI model at the time t for the parcel p
  - k is a constant factor
  - σ_t is the reliability of the model at the time t extracted from $\{\sigma_t\}_{t \in \underline{Y}}$

- Confidence assessment: the difference model(t; p, f_{t-1}) - VI(t) is related to the confidence level of the detection.

If the model of the unmowed grassland is not used, the mowing is detected applying a threshold on the decreasing of the current VI with respect to the last cloud-free sensed VI. These are the steps of the algorithm:

- Detection: A new grassland mowing is deteceted applying the AND of these two thresholding criteria:
  - VI(t) < VI(t_{cf}) – th_{VI}, where:
    - VI(t) is the current VI value
    - VI(t_{cf}) is the last cloud-free sensed VI
    - th_{VI} is the threshold used for the index VI
  - VI(t_{cf}) - VI(t) > R_{dec} N_{days}, where:
    - VI(t) is the current VI value
    - VI(t_{cf}) is the last cloud-free sensed VI
    - N_{days} is the number of days between t and t_{cf}
    - R_{dec} is the minimum allowed decreasing rate. Decreasing rates lower R_{dec} are assumed to be decreasing of the VI due to natural phenology of the grassland.
- Confidence assessment: the difference VI(t_{cf}) - VI(t) – th_{VI} is related to the confidence level of the detection.

2.4.3 Pseudo-Code

2.4.3.1 Library Import

 # standard Python library

```
import os, glob
import sys


import re
import configparser
import ast
import numpy as np
import dateutil.parser
import datetime
import shutil
from datetime import date
import scipy
import time


import gdal
```

```
from osgeo import osr, ogr
gdal.UseExceptions()
ogr.UseExceptions()
osr.UseExceptions()
import scipy.ndimage as ndimage
from scipy.signal import argrelextrema
```

 # new routines

```
import S2_gmd
import fusion
import model_lib
```

2.4.3.2 Tiles to be processed

```
tile_list = [T31UES, T31UET, T31UEU, T31UFS, T31UFT, T31UFU, T31UFV, T31UGS, T31UGT, T31UGU, T31UGV, T32ULB, T32ULC, T32ULD, T32ULE ] # S2 tiles ids covering the area of interest of Netherlands
```

2.4.3.3 Input Parameters

 # processing parameters

```
prod_type_list = _SNDVI_  # NDVI
invalid_val = -10000  # NDVI invalid value
sc_fact = 1000 # NDVI scaling factor
corrupted_th = 0.1 # NDVI below this value are considered not exploitable (bare soil, ploughing, …)
stat_smpl_n = 0 # number of images to calculate statistics
no_mowing_after_det = 60 # No mowing are expected before than 60 days
model_temporal_range_str = ('20180101 00:00:00', '20181231 23:59:59') # temporal range for the generation of the model (previous year wrt the current year of mowing detection)
process_temporal_range_str = ('20190401 00:00:00', '20191031 23:59:59') # temporal range for the detection
options_layer_burning = ['ALL_TOUCHED'] # touch option for GDAL
erode_pixels = 0 # pixel buffer to remove with morphological erosion
```

 # detection parameters, depending on the area/country

```
decreasing_abs_th = 0.05
```

2.4.3.4 Data and paths

```
S2DataRoot = <root directory where S2 derived data are stored>
segmentsFile = <name of the input shapefile used for segmentation>
outputShapeFile = <name of the output shapefile>
outputDir = <root directory where input preprocessed S2 data are stored>
```

2.4.3.5 File list generation and parameter extraction

 # generate list of file name to load for tile

```
    # setting file name parsing tools
    keys = ['file_name', 'data_type', 'acq_date', 'acq_time', 'tile_code']
    get_par_from_file = re.compile('(S2AGRI_L3B_([A-Z]{5,11})_A([0-9]{8})T([0-
9]{6})_(T[0-9]{2}[A-Z]{3}).TIF)')

    # file list generation from data dir
    file_list =
glob.glob(os.path.join(os.path.join(S2DataRoot,'*/*/*/*/'),'S2AGRI_*.TIF'))
    print(os.path.join(os.path.join(S2DataRoot,'*/*/*/*/'),'S2AGRI_*.TIF'))

    # extract data file names and dates from file list and for specific orbits
    par_list = S1_gmd_v2.read_file_list(file_list, get_par_from_file, keys,
tile_number, orbit_field_label='tile_code')

    # verify if there are data for specific tile
    if len(par_list) == 0:
        return 2

    # put filename and data parameters in a pandas df
    print("fill pandas structure")
    df = pd.DataFrame.from_records(par_list, columns=keys).drop_duplicates(keys[0])

    # date and time conversion from str
    df['acq_date_time' ] = pd.to_datetime(df.apply(lambda x:
x['acq_date']+'T'+x['acq_time'], axis=1), yearfirst=True, dayfirst=False)
```

2.4.3.6 Data selection based on dates and vegetation indexes

```
    # remove all vegetation indexes not required
    valid_prod_type_mask = False
    for pt in prod_type_list:
        valid_prod_type_mask = np.logical_or(valid_prod_type_mask, df['data_type']
== pt)
    df = df.loc[valid_prod_type_mask]

    # retrieval of the validity date/time range
    if older_acq_date:
        validity_temporal_range_str = [older_acq_date+"T000000",
new_acq_date+"T235959"]
    else:
        validity_temporal_range_str = [new_acq_date+"T000000",
new_acq_date+"T235959"]

    validity_temporal_range_str = [dateutil.parser.parse(date_str, yearfirst=True,
dayfirst=False) for date_str in validity_temporal_range_str]
    select_date_interval = [validity_temporal_range_str[0] -
datetime.timedelta(days=S2_time_interval*stat_smpl_n),
validity_temporal_range_str[1]]

    # File selection based on dates#
    valid_date_mask = (df['acq_date_time'] >= select_date_interval[0]) &
(df['acq_date_time'] <= select_date_interval[1])
    print(np.sum(valid_date_mask))
    df = df.loc[valid_date_mask]

    # generate lists of files and dates
    data_list = df['file_name'].values
```

2.4.3.7 Removal of corrupted files

```
    # remove corrupted files
    data_list = S1_gmd_v2.remove_corrupted_files(list(data_list))
```

2.4.3.8 Generation of segmentation map from input shapefiles and extent extraction

 # Extraction of projection of the input shapefile

```
    # Get projection from shape file
    ogr_data = ogr.Open(segmentsFile)
    Layer = ogr_data.GetLayer(0)
    spatialRef = Layer.GetSpatialRef()
    dst_srs = spatialRef.ExportToWkt()

    # Get shape file extent
    (min_x, max_x, min_y, max_y) = Layer.GetExtent()
    print("shape file extent", (min_x, min_y, max_x, max_y))

    # Some gdalwarp parameters
    resampling = gdal.GRA_Bilinear
    error_threshold = 0.125  # error threshold --> use same value as in gdalwarp
```

2.4.3.9 Generation of the GDAL Virtual Raster of the SAR data on the shapefile projection

```
    # make vrt of all data without considering extension of the shape file
    print('Make virtual raster of input data (without considering extension of the
shape file)')
    output_vrt_tmp = os.path.join(output_tmp_dir, "data_cube_tmp.vrt")
    images_n = S1_gmd_v2.make_vrt(data_list, dst_srs, output_tmp_dir,
output_vrt_tmp, outputBounds=None,
                                  srcNodata=invalid_data, resampling=resampling,
error_threshold=error_threshold)

    if images_n > 0:
        vrt_data = gdal.Open(output_vrt_tmp)
    else:
        print('Empty VRT. Does tile intersect segment layer?')
        #return 1
```

2.4.3.10 Extraction of the gdal virtual raster extent

```
    # Get virtual raster extension
    geoTr = vrt_data.GetGeoTransform()
    r_min_x = geoTr[0]
    r_max_y = geoTr[3]
    r_max_x = r_min_x + geoTr[1]*vrt_data.RasterXSize
    r_min_y = r_max_y + geoTr[5]*vrt_data.RasterYSize
    r_extent = (r_min_x, r_min_y, r_max_x, r_max_y)
    del vrt_data
```

2.4.3.11 Extraction of the intersection between gdal virtual raster extent and shapefile extent

```
    # Calculate extent intersection between virtual raster and shape file
    outputBounds = (np.maximum(min_x, r_min_x),
                    np.maximum(min_y, r_min_y),
                    np.minimum(max_x, r_max_x),
                    np.minimum(max_y, r_max_y))
```

2.4.3.12 Generation of the GDAL Virtual Raster of the SAR data on the shapefile projection and on the intersection extent

 # Finally a new gdal VRT is generated with the extent equal to the intersection between input shape file and S2 data available

```
    # make vrt over the intersection extent
    print('Make virtual raster of input data (considering intersection between
extensions of i- the shape file, ii- the virtual raster)')
    output_vrt = os.path.join(output_tmp_dir, "data_cube.vrt")
    images_n = S1_gmd_v2.make_vrt(data_list, dst_srs, output_tmp_dir, output_vrt,
outputBounds=outputBounds,
                                  srcNodata=invalid_data, resampling=resampling,
error_threshold=error_threshold)
    if images_n > 0:
        vrt_data = gdal.Open(output_vrt)
    else:
        print('Empty VRT. Does tile intersect segment layer?')
        #return 1
```

2.4.3.13 Generation of raster ROI mask and segmentation map from input shapefiles

 # Input segmentation shapefile is projected on the S2 data projection and cropped

 # Output segmentation raster file name building

```
segmentsOutputDir=os.path.join(outputDir, "segments")
segmentsOutRaster                  =                os.path.join(segmentsOutputDir,
os.path.basename(segmentsFile)[:-4]+'_raster')
```

 # make directories

```
try:
    os.mkdir(segmentsOutputDir)
except(OSError):
    print('directory ',segmentsOutputDir,' exists...moving on')
```

 # Generate segmentation raster map from input shapefile. The segmentation shapefile is rasterized over the raster imageNDVI_list[0] and write result on file

```
burned_pixels, seg_attributes = S1_gmd_v2.layer2mask(segmentsFile, output_vrt,
segmentsOutRaster, layer_type='segments', options=options_layer_burning)
```

2.4.3.14 Load data

```
    # load segments and truths
    gdal_data = gdal.Open(segmentsOutRaster)
    segments = gdal_data.ReadAsArray()
    segments_geo_transform = gdal_data.GetGeoTransform()
    segments_projection = gdal_data.GetProjection()
```

2.4.3.15 Morphological erosion of segments

 # Morphological erosion is applied to the segmentation mask if erode_pixels > 0

```
    eroded_segs = np.copy(segments)
    if erode_pixels > 0:
        print('Apply erosion')
```

```
        # first step: separate segments
        cross = np.array([[0,1,0],[1,1,1],[0,1,0]])
        min_seg = scipy.ndimage.minimum_filter(segments,footprint=cross)
        max_seg = scipy.ndimage.maximum_filter(segments,footprint=cross)
        segs_borders = (max_seg-min_seg)>0
        eroded_segs[segs_borders] = 0

        # second setp: erode
        if erode_pixels > 1:
            erosion_mask = scipy.ndimage.morphology.binary_erosion(eroded_segs>0,
structure=cross,iterations=(erode_pixels-1))
            eroded_segs = eroded_segs*erosion_mask
```

2.4.3.16 Extraction of segment id and their parameters

 # Extraction of the segments

```
unique_segments = np.unique(eroded_segs)

if unique_segments[0] == 0:

    # 0 label corresponds to not valid segments and it will be removed

    unique_segments = unique_segments[1:]
```

 # calculate pixels within segments

```
seg_pixels_num     =     np.array(scipy.ndimage.sum(eroded_segs>0,     eroded_segs,
index=unique_segments))# calculate center of the segments
```

2.4.3.17 Extraction data parameters from gdal virtual raster

```
    # extract data parameters and dates from file_list in the vrt data
    par_list = S1_gmd_v2.read_file_list(vrt_data.GetFileList()[1:],
get_par_from_file, keys, tile_number, orbit_field_label='tile_code')

    # put filename and data parameters in a pandas df
    vrt_df = pd.DataFrame.from_records(par_list, columns=keys)

    # date and time conversion from str
    vrt_df['acq_date_time'] = pd.to_datetime(vrt_df.apply(lambda x:
x['acq_date']+'T'+x['acq_time'], axis=1), yearfirst=True, dayfirst=False)
```

2.4.3.18 Feature Extraction based on segmentation

 # selection of the average method

```
    #selection of the average method
    stat_p = scipy.ndimage.mean

    # Write list of df
    list_df = S1_gmd_v2.load_stats(vrt_data, vrt_df, eroded_segs, unique_segments,
seg_attributes, seg_parcel_id_attribute, stat_p, invalid_data)

    # Concatenate all pandas elements in the list
    data_df = pd.concat(list_df, ignore_index=True)

    # Sort wrt count
    data_df.sort_values(['count'], ascending=[False], inplace=True)

    # Aggregate with respect the same date and same parcel_id by keeping the stats
associated with the highest count
```

```
#    data_df = data_df.loc[data_df.groupby(['acq_date', 'data_type',
seg_parcel_id_attribute])['count'].aggregate('idxmax')]
    data_df = data_df.groupby(['acq_date', 'data_type',
seg_parcel_id_attribute]).aggregate('first')

    # Rearrange structure to put dates and data_types as columns
    data_df = pd.pivot_table(data_df, values=['mean','count'],
index=[seg_parcel_id_attribute],
                             columns=['data_type', 'acq_date'], dropna=False)

    # Sort with respect the dates (from early to late)
    data_df = data_df.sort_values(['acq_date'], axis=1, ascending=True)
```

2.4.3.19 Extraction of the temporal series

 # Extraction of temporal series from pandas

```
    parcel = data_df.index.values

    # generate array with temporal series
    VI_seg = []
    for pt, sf, ct in zip(prod_type_list, sc_fact, corrupted_th):
        VI_data = data_df.xs(key=('mean', pt), level=(0, 'data_type'),
axis=1).values
        dates_str = data_df.xs(key=('mean', pt), level=(0, 'data_type'),
axis=1).columns.values
        print(dates_str)
        VIDateList = [dateutil.parser.parse(d, yearfirst=True, dayfirst=False) for
d in dates_str]
        VI_seg.append({'data_type': pt, 'date_list': VIDateList, 'VI': VI_data,
'sc_fact': sf, 'corrupted_th': ct})

    # apply scale factors and remove corrupted VI
    for d in VI_seg:
        d['VI'] /= d['sc_fact']
        d['VI'][ d['VI'] < d['corrupted_th'] ] = np.nan


    # Final temporal serie
    NDVI_seg = np.array(VI_seg[0]['VI'])
    NDVIDateList = VI_seg[0]['date_list']
```

2.4.3.20 Make dictionary of the segments and its inversion

```
    # make dictionary segment --> seg_parcel_id_attribute
    seg_attributes = {i: {seg_parcel_id_attribute: p} for i, p in
enumerate(parcel)}

    # inverti dizionari
    inv_seg_dct = {v[seg_parcel_id_attribute]:k for k,v in seg_attributes.items()}
```

2.4.3.21 Model extraction

 # Extraction of the model based on data from previous years

```
    # definition of the models
    NDVI_nomow_model = np.nanpercentile(NDVI_seg, NDVI_nomow_model_perc, axis=0).T
    NDVI_mean = np.nanmean(NDVI_seg, axis=0).T
    NDVI_std = np.nanstd(NDVI_seg, axis=0).T
    parcel_n = np.sum(NDVI_seg>0, axis=0).T # number of parcels
```

```
    # valid indexes
    val_ind = np.where(np.logical_and(NDVI_nomow_model > min_val_VI, (parcel_n >
p_n_th)))[0]

    # raw model
    x = doy[val_ind]
    y = NDVI_nomow_model[val_ind]
    n = parcel_n[val_ind]

    # double_logistic fit: parameters
    # The starting point to initialize the constrained minimization and the
constrains are set
    # start_params = [double_logistic pedestal (min), amplitude, rise speed, rise
inflection time, fall speed, fall inflection time]
    # bound = ([lower bound list],[upper bound list])

    start_params = [y.min(), y.max()-y.min()] + list(start_params)
    bounds = ([0., 0.] + list(bounds[0]), [np.inf, np.inf] + list(bounds[1]))

    # model fitting
    res = pheno_func.constrained_fit_phenology_model(
                x, y, pheno_model = "dbl_logistic", params=start_params,
bounds=bounds)

    dbl_log_params_fit = res.x
    res_cost = res.cost
    res_opt = res.optimality
    message = res.message

    # double_logistic interpolation on dense regular grid
    x_mod = np.linspace(1., 366, num=int(366/sampling_days))
    model_dbl_l = pheno_func.get_model(x_mod, None, params=dbl_log_params_fit,
pheno_model = "dbl_logistic")
```

2.4.3.22 Initialization

 # Allignement of model x's to the current year

```
    # Fit of the model to the available DOYs
    x = model_dict['SNDVI'][:,0]
    model_dbl_l = model_dict['SNDVI'][:,1]

    # final no-mowing model
    NDVI_nomow_model_ = np.interp([d.timetuple().tm_yday for d in NDVIDateList], x,
model_dbl_l)

    # Adapt/initialize no-mowing model to each parcel
    print("Adapt/initialize no-mowing model to each parcel")
    NDVI_nomow_model = np.zeros_like(NDVI_seg)
    for i in range(len(NDVI_seg)):
        indx = np.where(np.isfinite(NDVI_seg[i,:]))[0]
        if np.size(indx) > 0:
            indx = indx[0]
            NDVI_nomow_model[i,:] =
NDVI_nomow_model_*(NDVI_seg[i,indx]/NDVI_nomow_model_[indx])
```

2.4.3.23 Detection

The detection is driven by the parameter apply_model. If apply_model = True, for the detection is used the model of the unmowing grassland, otherwise the detection is performed considering the VI decreasing with respect to the last cloud-free VI.

 # Perform the mowing detection

```
If apply_model == True:

    # Detection on model
    NDVI_det_cube = np.zeros_like(NDVI_seg)
    for t in np.arange(1, NDVI_det_cube.shape[1]):
        # detection at t
        NDVI_det_cube[:,t] = np.maximum((NDVI_nomow_model[:,t] - NDVI_seg[:,t] -
decreasing_abs_th)/NDVI_nomow_model[:,t-1], 0)  # C1: decremento assoluto wrt model
        # only for the parcel with detected mowing or with an increasing of the VI,
the model is updated
        to_update_parc = list(np.where(NDVI_det_cube[:,t] > 0)[0])
        increased_NDVI = list(np.where(diff_seg[:,t] > 0)[0])
        to_update_parc = np.array(to_update_parc + increased_NDVI)
        if len(to_update_parc) > 0:
            NDVI_nomow_model[to_update_parc, t:] *= (NDVI_seg[to_update_parc,
t]/NDVI_nomow_model[to_update_parc, t])[:,None]

    det_cube = NDVI_det_cube
    det_cube[np.isnan(det_cube)] = 0

If apply_model != True:

    for t in range(NDVI_det_cube.shape[1]):

        # Detection at t
        # Detection of a mowing if the decreasing angle (alpha =
atan(delta_VI/delta_doy)) is enough small, that is lower than decreasing_rate_th.
(VI decreasing enough fast.)

        NDVI_det_cube[:,t] = np.maximum((NDVI_last - NDVI_seg[:,t] -
decreasing_abs_th)/NDVI_last, 0)             # VI decreasing is at least
decreasing_abs_th
        NDVI_det_cube[:,t] *= ((NDVI_seg[:,t] - NDVI_last) / (DOYList[t] -
NDVI_last_doy) < decreasing_rate_th)   # detection only if the decreasing is enough
fast

        # update always NDVI_last and NDVI_doy and NDVI_last_hist

        if t == 0:
            NDVI_last_hist[:,t] = NDVI_last
        elif t > 0:
            NDVI_last_hist[:,t] = NDVI_last_hist[:,t-1]
        valid_vi_indexes = list(np.where(np.isfinite(NDVI_seg[:,t]))[0])
        if len(valid_vi_indexes) > 0:
            NDVI_last[valid_vi_indexes] = NDVI_seg[valid_vi_indexes,t]
            NDVI_last_doy[valid_vi_indexes] = DOYList[t]
            NDVI_last_hist[valid_vi_indexes, t] = NDVI_seg[valid_vi_indexes,t]
```

2.4.3.24 Remove multiple detections

 # Remove multiple detections based on "first ones win" and get maximum confidences

```
    for i in np.arange(1,det_cube.shape[1]):
        print("iter   : ", i)
        first_indx = max(np.searchsorted(NDVIDateList, NDVIDateList[i] -
datetime.timedelta(days=no_mowing_after_det)),0)
        det_mow_indx = np.where(det_cube[:, i]>0)[0]
        print("det_mow_indx", det_mow_indx)
        print(np.size(det_mow_indx))
        if np.size(det_mow_indx) > 0:
```

```
            print(first_indx, i)
            prev_det_indx = np.where(np.max(det_cube[det_mow_indx, first_indx:i],
axis=1)>0)[0]
            print("prev_det_indx", prev_det_indx)
            if np.size(prev_det_indx) > 0:
                print("det_mow_indx[prev_det_indx]", det_mow_indx[prev_det_indx])
                det_cube[det_mow_indx[prev_det_indx], i] = 0
```

2.4.3.25 Confidence normalization

 # The S2 detection confidence are normalized in the range (0.5, 1.0)

```
alpha = 1.0
det_cube[det_cube>0]       =       S2_gmd.norm_fun(det_cube[det_cube>0],       alpha,
bounds=(0.5,1.0))
```

2.4.3.26 Write detection shape file results

 # If outputShapeFile does not exist, it is assumed that the input segmentsFile can be used to derive geometry and it is cloned on the output directory

```
if not os.path.exists(outputShapeFile):
    fusion.cloneAndUpdateShapefile(segmentsFile, outputShapeFile)
```

 # The new detections are added in the output shape file outputShapeFile

```
fusion.writeDetections_S2(outputShapeFile, unique_segments, det_cube, NDVIDateList,
np.isfinite(NDVI_seg), "S2", minimum_interval_days=30)
```

2.4.3.27 Calculate compliancy

 # Run compliancy calculation (for details see section 2.5)

```
        fusion.do_compliancy(outputShapeFile, cnt_crop_code, cnt_crop_TR,
cnt_crop_rule)
```

2.5 Fusion of detections

2.5.1 S-1/S-2 mowing detections merge

For each new S-1 or S-2 (related VI) image, a new detection processing is performed and the results are fused with the previous ones, according to the following methodology:

- By default, give to S2 a confidence level (within (0.5, 1)) always higher than S1 (0, 0.5);
- For each parcel, a new detection is included if:
  - its confidence is in the top 4 most confidence detections and;
  - it has a temporal distance higher than n days with respect to the most confident detections.

For the pseudo-code, refer to 2.2.3.25.

2.6 Compliancy assessment

2.6.1 Compliancy criteria for each country

The compliancy of the mowing events detected is assessed taking in account the national regulations provided by the Paying Agencies. Depending on regulations, compliancy assumes a value from 0 to 2 or from 0 to 3:

- "0": Not assessed (if proc attribute of the <OutputShapeFile> =0)
- "1": Assessed and compliant because a mowing occurred in the reference period
- "2": Assessed and not compliant because no mowing occurred in the reference period
- "3": Assessed and not compliant because a mowing occurred in the reference period and also outside the reference period

Table 2-4 to Table 2-9 define for each country and for each grassland crop type, the compliancy rules. Only for Lithuania, there are different rules depending on crop type.

Table 2-4. Czech Republic grassland mowing regulations

Country: Czech Republic			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• 315, Temporary Grassland • 350, Permanent Grassland • 3001, Permanent Grassland	At least 1 mowing between 1st April and 31st October	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the mandatory period even if a mowing occurred outside the mandatory period	Compliancy domain [0, 1, 2] • "0": Not assessed • "1": Assessed and compliant because a mowing occurred in the reference period "2": Assessed and not compliant because no mowing occurred in the reference period

Table 2-5. Italy grassland mowing regulations

Country: Italy			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• 46, Loietto Loglio • 51, Lupolina • 65, Grassland • 79, Vetch • 152, Clover • 336, Meadow • 389, Vetch species • 390, Vetch species • 460, Grassland with orchid • 461, Herbal species • 562, Alpha-Alpha • 581, Annual grassland • 612, Annual grassland • 800, Annual grassland • 840, Annual grassland • 862, Fenugreek • 899, Permanent grassland	At least 1 mowing or grazing between 1st April and 31st October	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the mandatory period even if a mowing occurred outside the mandatory period	Compliancy domain [0, 1, 2] • "0": Not assessed • "1": Assessed and compliant because a mowing occurred in the reference period • "2": Assessed and not compliant because no mowing occurred in the reference period

Table 2-6. Lithuania grassland mowing regulations

Country: Lithuania			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• GPŽ, Pasture or meadow, perennial grass up to 5 years	At least 1 mowing or grazing within 31st July	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the	Compliancy domain [0, 1, 2] • "0": Not assessed • "1": Assessed and compliant because a mowing occurred in the reference period "2": Assessed and not compliant because no
• DGP, Perennial pastures or meadows 5 years and more			
• GPA, Pasture or meadow, perennial grass up to 5 years, renewed in the current year			

• EPT, Extensive meadows grazing with livestock	At least 1 grazing between 1st May and 30th October	mandatory period even if a mowing occurred outside the mandatory period	mowing occurred in the reference period
• SPT, Specific meadows	At least 1 mowing between 15th July and 15th October		
• 5PT-2, Extensive management of wetlands (direct payments are paid)	At least 1 mowing between 15th July and 1st March (next year)		
• MNP, Aquatic warbler habitats storage in raw and semi-natural grasslands	At least 1 mowing or grazing between 1st July and 1st October		
• MNS. Aquatic warbler habitats storage in wetlands	At least 1 mowing or grazing between 1st August and 1st October		

Table 2-7. Netherlands grassland mowing regulations

Country: Netherlands			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• 265, Grassland, permanent • 266, Grassland temporarily • 331, Grassland, natural Main function of agriculture • 332, Grassland natural. Main function nature • 333, Edge adjacent to permanent pasture or permanent cultivation consisting mainly of permanent grass	At least 1 mowing between 1st April and 31st October	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the mandatory period even if a mowing	Compliancy domain [0, 1, 2] • "0": Not assessed (if proc = 0) • "1": Assessed and compliant because a mowing occurred in the reference period "2": Assessed and not compliant because

• 334, Edge adjacent to land mainly consisting of permanent grass. (EA: not managed) • 370, Edge adjacent to permanent pasture or permanent cultivation mainly consisting of temporary grass • 372, Edge adjacent to land mainly consisting of temporary grass. (EA: not managed) • 383, Grass seed • 1921, Grass sods • 3506, English raaigrass • 3509, Festulolium • 3512, Italianraaigrass • 3513, Westerwoldsraaigrass • 3519, Sorghum • 3522, Timothee • 3523, Veldbeemdgras • 3805, Rietzwenkgras industrial grass • 3807, Rietzwenkgras other than for industrial grass • 3808, Roodzwenkgras		occurred outside the mandatory period	no mowing occurred in the reference period

Table 2-8. Spain grassland mowing regulations

Country: Spain			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• 2, Alfalfa • 85, Grassland pasture	At least 1 mowing or grazing between 1st April and 31st October	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the mandatory period even if a mowing occurred outside the mandatory period	Compliancy domain [0, 1, 2] • "0": Not assessed (if proc = 0) • "1": Assessed and compliant because a mowing occurred in the reference period • "2": Assessed and not compliant because no mowing occurred in the reference period

Table 2-9. Romania grassland mowing regulations

Country: Romania			
Crop type (code, name)	**Mandatory mowing period**	**Additional rules in case of mowing event is outside the mandatory period**	**Compliancy domain**
• 450, Temporary grassland (artificial, sowed on AL < 5 years) • 603, Public permanent grasslands used in common • 604, Permanent grasslands used in common • 605, Public permanent grasslands used individually • 606, Permanent grasslands used individually • 607, Individually used meadows • 608, Public meadows used individually • 609, Pasture individual • 610, Pasture shared • 611, Hay shared mowed	At least 1 mowing between 1st May and 31st October	Rules: 1) Compliant (C) if a mowing event occurred in the mandatory period and also outside the mandatory period 2) Not compliant (NC) if no mowing occurred in the mandatory period even if a mowing occurred outside the mandatory period	Compliancy domain [0, 1, 2] • "0": Not assessed (if proc = 0) • "1": Assessed and compliant because a mowing occurred in the reference period • "2": Assessed and not compliant because no mowing occurred in the reference period

• 612, Pasture communal but used individually • 660, Traditional orchard extensively used by pasturage and or mowing • 661, Traditional orchard extensively used as meadow • 662, Energy natural meadows • 663, Orchard traditional extensive pasture • 671, Convert sensitive PP			

2.6.2 Pseudo-code

2.6.2.1 Library Import

 # standard library

```
import os
import sys
import numpy
import datetime
import time
```

2.6.2.2 Parameters

 # each country (Czech Republic (cz), Italy (it), Lithuania (lt), Netherlands (nl), Romania (ro) and Spain (sp)), France (fr) has different crop types

```
cnt=[cz, it, lt, nl, ro, sp]
cnt_crop_id_dict = [“crop_code_1”: 0, “crop_code_2”: 1, ..., “crop_code_n”=n-1]
```

 # each country has, for each crop type, own temporal ranges as mandatory mowing period

 # List of tuples. Each 2-elements tuple contain start and end dates of the mandatory mowing period for each crop type

```
cnt_time_range =
[(datetime.datetime.strptime('dd/mm/yyyy',                    "%d/%m/%Y"),
datetime.datetime.strptime('dd/mm/yyyy, "%d/%m/%Y")),  #for crop_code_1
(datetime.datetime.strptime('dd/mm/yyyy                ',     "%d/%m/%Y"),
datetime.datetime.strptime('dd/mm/yyyy’, "%d/%m/%Y")), #for crop_code_2
, ...,
(datetime.datetime.strptime('dd/mm/yyyy                ',     "%d/%m/%Y"),
datetime.datetime.strptime('dd/mm/yyyy’, "%d/%m/%Y")), #for crop_code_n]
```

2.6.2.3 Compliancy calculation

```
if cnt = “cz” or cnt = “it” or cnt = “nl” or cnt = “ro” or cnt = “sp”
    # iteration over the elements of grass_prod (parcels)
    for parcel in grass_prod:    # grass_prod is an NxM array containing the N lines of the dbf of the output shapefile <outputShapeFile> for the selected country
        t_range=cnt_time_range [cnt_crop_id_dict[get crop code (parcel)] #t_range is the start and end dates of the mandatory mowing period for the parcel and crop
        t_start = t_range[0]
        t_end = t_range[1]
```

 # cond1: if the parcel has not mowing (mow_n attribute of the <outputShapeFile>= 0) and has not been processed (proc attribute of the <outputShapeFile>= 0) then compliancy = 0 (Not assessed)

```
        if parcel[mow_n] = 0 and parcel[proc] = 0:
            compliancy=0
```

 # cond2: if the parcel has not mowing (mow_n attribute of the <outputShapeFile>= 0) and has been processed (proc attribute of the <outputShapeFile>= 0) then compliancy = 2 (Assessed and not compliant because no mowing)

```
        if parcel[mow_n] = 0 and parcel[proc] = 1:
            compliancy =2
```

 # cond3: if the parcel has n mowing (mow_n attribute of the <outputShapeFile> > 0 (values from 1 to 4) and, at least, a mowing intersects the mandatory mowing period for the parcel and crop, then compliancy=1 (Assessed and compliant) otherwise compliancy=2 (Assessed and not compliant because no mowing)

```
        if parcel[mow_n] > 0
            mowing_in_interval = False
            for mow_event in range(1, mow_n):
      if (parcel[m<mow_event>_start] >= t_start and parcel[m<mow_event>_start] <= t_end) or parcel[m<mow_event>_end] >= t_start and parcel[m<mow_event>_end] <= t_end):
                    mowing_in_interval = True
                    continue
            if mowing_in_interval:
                compliancy = 1
            else:
                compliancy = 2
    compliancy → parcel[compliancy]  # the compliancy value is written back in the dbf file
else:
```

 # iteration over the elements of grass_prod (parcels)

```
    for parcel in grass_prod:    # grass_prod is an NxM array containing the N lines of the dbf of the output shapefile <outputShapeFile> for the selected country
    t_range=cnt_time_range [cnt_crop_id_dict[get crop code (parcel)] #t_range is the start and end dates of the mandatory mowing period for the parcel and crop
        t_start = t_range[0]
```

```
t_end = t_range[1]
```