Sentinel-2 Agriculture Vegetation status DPM

Jordi Inglada

Sentinel-2 Agriculture

Vegetation status DPM

Contents

1 Introduction

This document describes the proposed processing chain for the production of the vegetation status products for the Sentinel-2 Agriculture project.

The algorithm description and justification of choices have been documented in the Design Justification File[1]. The present document describes the processing chain and its subsystems.

Where possible, we use standard components available in the Orfeo Toolbox version 4.4[2]. When no equivalent component is available in the Orfeo Toolbox, the algorithm is described using either pseudo-code or the example implementations in Python and C++ available at `http://tully.ups-tlse.fr/jordi/otb-bv` (LAI retrieval and profile reprocessing) and `http://tully.ups-tlse.fr/jordi/phenotb` (NDVI temporal metrics).

2 System components

2.1 LAI retrieval

2.1.1 Overview of the subsystem

The LAI retrieval is performed by using machine learning to build a non-linear regression model. The regression model is estimated using simulated satellite reflectances. These reflectances are simulated using the ProSail model. This approach is based on the LAI procedure S2PAD[3], the main difference being that the model is applied to every acquisition date, which allows removing the solar and sensor angles from the predictor variables. Figure 1 illustrates this procedure.

After that, 2 reprocessing options are available in order to improve the LAI retrieval by taking into account the multi-temporal information. The first option is an on-line algorithm which uses the n last acquisitions to estimate the LAI value for the last one. It uses the error estimation of the LAI retrieval to weight the LAI values. The second option can be applied at the end of the season and consists in fitting a phenological model (double logistic function) to the LAI retrieval time series. Figure 2 illustrates the multi-temporal retrieval approaches.

The LAI retrieval processing chain is implemented as a set of applications corresponding to the individual processing blocks. These applications are based on a C++ library which relies on the Orfeo Toolbox (see section 3.1). A Python module is also provided in order to better integrate the applications into a processing chain.

2.1.2 The applications

There are 6 applications used to implement the LAI retrieval processing chain.

1. `BVInputVariableGeneration`: generates a distribution of input variables for the ProSail model which have the correct statistics and correlations.
2. `ProSailSimulator`: simulates satellite reflectances using the ProSail variables, the acquisition configuration and the satellite spectral bands.
3. `InverseModelLearning`: uses the simulated reflectances for each input LAI in order to build a regression model. This application can also be used to build a regression model for the LAI retrieval error.
4. `BVInversion`: application which applies the regression models to a set of input reflectances. It is mainly used for validation purposes.
5. `BVImageInversion`: like `BVInversion` but applied to full images.
6. `ProfileReprocessing`: application which applies the multi-temporal algorithms to an LAI time series.

In the following sections we show how to use these applications from the command line.

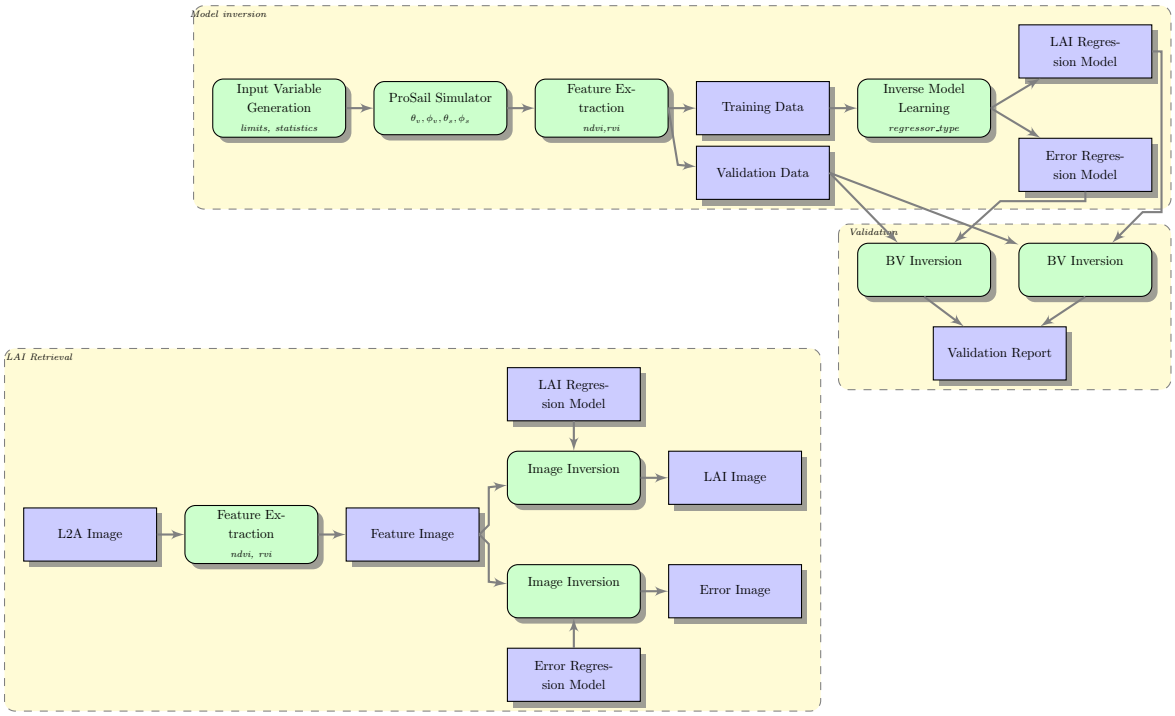

Figure 1: Block diagram for the mono-date LAI retrieval procedure.

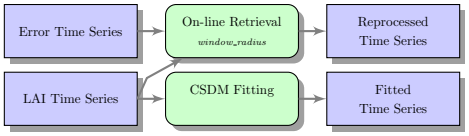

Figure 2: Block diagram for the 2 multi-date LAI retrieval procedures.

2.1.2.1 Input Variable Generation

Table 1 lists the variables used by the `BVInputVariableGeneration` application. The application is used as follows:

```
otbcli_BVInputVariableGeneration -samples $samples -out $outfile
```

The statistical distribution of the variables is described in table 2. The values are drawn using the corresponding probability density defined by its mean and standard deviation (except for the uniform density which is defined by its minimum and maximum). For the Gaussian and lognormal densities, the values are kept only if they are within the range defined by the minimum and the maximum. Finally, in order to take into account the inherent correlation between LAI and the other variables, the following transformation is applied to each sample for each variable:

[1]Sentinel-2 for Agriculture Design Justification File (v1.1, 2015/04/15)
[2]This version corresponds to the changeset http://hg.orfeo-toolbox.org/OTB/file/baf740ee2113.
[3]S2PAD-VEGA-ATBD-0003-2$_{1\text{L2BATBD}}$. S2PAD - Sentinel-2 MSI - Level 2B Products Algorithm Theoretical Basis Document. Issue 2.1. 2010/10/13.

Table 1: Variables for the input variable generation

Input variable	role	default value
`samples`	Number of samples to be generated	40000
Output variable	role	
`outfile`	Name of the ASCII file to store the samples	–

$$V* = V_{mean} + (V - V_{mean}) * (LAI_{Conv} - MLAI)/LAI_{Conv}$$

This configuration is taken from [3] and slightly modified to match recent evolutions of the BV-NET software implemented by INRA (mainly in terms of switching from a Gaussian distribution to a lognormal one for the LAI).

The values of these parameters may evolve as a detailed analysis of their representativity over the Sentinel-2 Agriculture test sites is ongoing.

Table 2: Ranges and parameters for the statistical distribution of the input variables

	Variable	Minimum	Maximum	Mean	Std	Law	LAI_{Conv}
Canopy	MLAI	0.0	5.0	0.5	1.0	lognormal	–
	ALA (°)	5	80	40	20	Gaussian	10
	$Crown_{Cover}$	1.0	1.0	–	–	uni	10
	HsD	0.1	0.5	0.2	0.5	Gaussian	1000
Leaf	N	1.20	1.80	1.50	0.30	Gaussian	10
	Cab (μ g.m -2)	20	90	45	30	Gaussian	10
	Cdm (g.m-2)	0.0030	0.0110	0.0050	0.0050	Gaussian	10
	Cw_{Rel}	0.60	0.85	–	–	uni	10
	Cbp	0.00	2.00	0.00	0.30	Gaussian	10
Soil	Bs	0.50	3.50	1.20	2.00	Gaussian	10

Table 3: Variables for the ProSail simulation

Input variable	role	default value
`bvfile`	File containing the ProSail variables	–
`rsrfile`	File containing the satellite spectral responses	–
`solarzenith`	Solar zenithal angle	–
`sensorzenith`	Sensor zenithal angle	–
`azimuth`	Relative azimuth between sensor and Sun	–
`noisevar`	Noise variance to be added to the reflectances	0.01
Output variable	**role**	
`out`	Name of the ASCII file to store the simulated reflectances	–

2.1.2.2 ProSail Simulator Table 3 lists the variables used by the `ProSailSimulator` application. The application is used as follows:

```
otbcli_ProSailSimulator -bvfile $bvFile -rsrfile $rsrFile -out $outputFile\
                        -solarzenith $solarZenithAngle\
                        -sensorzenith $sensorZenithAngle\
                        -azimuth $solarSensorAzimuth\
                        -noisevar $noisevar
```

2.1.2.3 Inverse Model Learning Table 4 lists the variables used by the `InverseModelLearning` application. The first column of the training file corresponds to the dependent variable and the following columns are the predictors. Added to the reflectances, the NDVI and the RVI (clamped to 30) are used as predictors. The normalization file contains 2 columns corresponding to the minimun and maximum values to be used for the normalization of each variable. Each row corresponds to a variable in the same order of the columns of the training file.

The application is used as follows:

Table 4: Variables for the inverse model learning

Input variable	role	default value
`training`	File containing the input/output pairs	–
`regression`	Regression algorithm: svr, nn, mlr	nn
`normalization`	File containig the min and max for each variable	–
Output variable	role	
`out`	Name of the file containing the regression model for the variable	–
`errest`	Name of the file containing the regression model for the error estimation	–

```
otbcli_InverseModelLearning -training $trainingFile -out $modelfile\
                            -errest ${modelfile}_errest\
                            -regression svr\
                            -normalization $normalizationFile\
                            -bestof 1
```

2.1.2.3.1 Note on spectral bands for Sentinel-2 Only bands at 10 m and 20 m resolution will be used as predictors.

The blue band (B2) will not be used due to residual atmospheric effects. B8 will not be used because of its overlap with B7 anb B8a. Therefore, B3, B4, B5, B6, B7, B8a, B11 and B12 will be used.

2.1.2.3.2 Note on the use of vegetation indices as predictors The use of vegetation indices as predictors has to be allowed as indicated above, but will be desactivated by default in the system.

2.1.2.4 Inversion There are 2 applications for the inversion. The first one operates on ASCII files as those used by the `InverseModelLearning` and is used for validation purposes. The second one operates on images and is used for the LAI retrieval.

It is worth noting that the same application is used for the LAI retrieval and for the error estimation, the only difference being the regression model provided to it.

Table 5: Variables for the variable inversion on ASCII files

Input variable	role
`reflectances`	File containing reflectances
`model`	Regression model file
`normalization`	File containig the min and max for each variable
Output variable	role
`out`	Name of the file containing the variable retrieval

2.1.2.4.1 Inversion from ASCII files Table 5 lists the variables used by the `BVInversion` application. The application is used as follows:

```
otbcli_BVInversion -reflectances $reflectanceFile -model $modelFile\
                   -normalization $normalizationFile\
                   -out $outputFile
```

2.1.2.4.2 Image Inversion Table 5 lists the variables used by the `BVImageInversion` application. The application is used as follows:

```
otbcli_BVImageInversion -in $inputimage -model $modelname \
                        -out $laiimage -normalization $normname
```

Table 6: Variables for the variable inversion using images

Input variable	role
`in`	Input image
`model`	Regression model file
`normalization`	File containig the min and max for each variable
Output variable	role
`out`	Name of the image file containing the variable retrieval

Table 7: Variables for the profile reprocessing

Input variable	role	default value
`ipf`	Input time profile	–
`algo`	Reprocessing algorithm: local or fit	local
`algo.local.bwr`	Backward radius of the window for the local algorithm	3
`algo.local.fwr`	Forward radius of the window for the local algorithm	0
Output variable	role	
`opf`	Output profile	–

2.1.2.5 Profile Reprocessing

Table 7 lists the variables used by the `ProfileReprocessing` application.

It is worth noting that the current version of the `otb-bv` package does not have a version of this application which is able to operate on image time series and the input and output profiles are ASCII files containing the values for each date for individual pixels. Adapting the application to operate on images is trivial by using the `itk::UnaryFunctorImageFilter` and passing it the appropriate function defined in `otbProfileReprocessing.h`. See section 3.1.5 for details.

2.1.2.5.1 On-line reprocessing

In the case of the on-line reprocessing, the application is used as follows:

```
otbcli_ProfileReprocessing -ipf $inputprofile\
                           -opf $outputprofile\
                           -algo local\
                           -algo.local.bwr 2\
                           - algo.local.fwr 0
```

In this case, the choice of radiuses implies that the last date is reprocessed ($fwr = 0$) using also the 2 previous dates ($bwr = 2$).

In general, one has to take into account the fact that the number of previous dates needed for the reprocessing may not be available. Therefore, the following conditions will be implemented:

- if only the current date is available (without no other backward image) then the current date value is used;
- if only the current date and only $bwr - 1$ (one in the default case) backward dates are available, then only the current date is used and the backward ones are actually ignored; this means that $bwr + 1$ valid dates, including the current one are needed to perform the reprocessing;
- if the current date and $N \geq bwr$ previous dates are available, then the value is reprocessed using all of them; using all dates simplifies the code and given the weight used for all dates, it does not modify the result in a significative manner.

2.1.2.5.2 CSDM fitting

In the case of the fitting reprocessing, there are no specific parameters and the application is used as follows:

```
otbcli_ProfileReprocessing -ipf $inputprofile\
                           -opf $outputprofile\
                           -algo fit
```

2.1.3 The Python module

The Python module offers a wrapper around the applications to facilitate the implementation of a complete processing chain. This wrapper is not limited to the Python wrappers automatically generated for the applications. It also offers pre- and post-processing steps.

We start by defining a set of indices and names for the PROSPECT and Sail variables. The indices here have to be coherent with the order used in the bv file and the definition of the vars in `otbBVTypes.h`.

```
bvindex = {"MLAI": 0, "ALA": 1, "CrownCover": 2, "HsD": 3, "N": 4, "Cab": 5, \
           "Car": 6, "Cdm": 7, "CwRel": 4, "Cbp": 9, "Bs": 10, "FAPAR": 11, \
           "FCOVER": 12}
bv_val_names = {"MLAI": ['gai', 'lai-bvnet'], "FAPAR": ['fapar', 'fapar-bvnet'],\
                "FCOVER": ['fcover', 'fcover-bvnet']}
```

A simple wrapper for the `BVInputVariableGeneration` application is used to generate the draws of the PROSPECT and Sail variables (see section 2.1.2.1):

```
def generateInputBVDistribution(bvFile, nSamples):
    app = otb.Registry.CreateApplication("BVInputVariableGeneration")
    app.SetParameterInt("samples", nSamples)
    app.SetParameterString("out", bvFile)
    app.ExecuteAndWriteOutput()
```

The generation of the training data for the inversion uses a wrapper for the `ProSailSimulator` application and some post-processing of the generated files in order to add vegetation indices, viewing angles[4], etc. as predictor variables. The code is also able to selec FCover or FAPAR as dependent variables, but this option is not relevant to the current version of the Sentinel-2 Agriculture processing chains.

```
def generateTrainingData(bvFile, simuPars, trainingFile, bvidx,
                         add_angles=False, red_index=0, nir_index=0):
    """
    Generate a training file using the file of biophysical vars (bvFile) and
    the simulation parameters dictionary (simuPars).
    Write the result to trainingFile. The first column will be the biovar to
    learn and the following columns will be the reflectances.
    The add_angles parameter is used to store the viewing and solar angles as
    features. If red_index and nir_index are set, the ndvi and the rvi are also
    used as features. red_index=3 means that the red reflectance is the 3rd
    column (starting at 1) in the reflectances file.
    """
    app = otb.Registry.CreateApplication("ProSailSimulator")
    app.SetParameterString("bvfile", bvFile)
    app.SetParameterString("soilfile", simuPars['soilFile'])
    app.SetParameterString("rsrfile", simuPars['rsrFile'])
    app.SetParameterString("out", simuPars['outputFile'])
    app.SetParameterFloat("solarzenith", simuPars['solarZenithAngle'])
    app.SetParameterFloat("sensorzenith", simuPars['sensorZenithAngle'])
    app.SetParameterFloat("azimuth", simuPars['solarSensorAzimuth'])
    app.SetParameterFloat("noisevar", simuPars['noisevar'])
    app.ExecuteAndWriteOutput()
    #combine the bv samples, the angles and the simulated reflectances for
    #variable inversion and produce the training file
    with open(trainingFile, 'w') as tf:
        with open(bvFile, 'r') as bvf:
```

[4] Although sensor and solar angles can be used as predictors, we have made the choice of using a different regression model per viewing configuration and therefore the angles are not used as predictors. This implies that the reflectance simulations have to be performed for every viewing configuration setting the angles correspondingly. Since Sentinel-2 products will contain angular grids, it is possible to implement different models within a single scene. However, no sensitivity analysis has been performed allowing to assess the usefulness of this approach. Since this does not introduce much complexity into the system, a tiling scheme within scenes taking into account a fixed angular step should be implemented.

```
            bvf.readline() #header line
            with open(simuPars['outputFile'], 'r') as rf:
                #the output line follows the format:
                #outputvar inputvar1 inputvar2 ... inputvarN
                for (refline, bvline) in zip(rf.readlines(), bvf.readlines()):
                    outline = ""
                    if bvidx == bvindex["FCOVER"] :
                        outline = string.split(refline)[-1]
                    elif bvidx == bvindex["FAPAR"] :
                        outline = string.split(refline)[-2]
                    else:
                        outline = string.split(bvline)[bvidx]
                    outline = outline+" "+\
                              string.join(string.split(refline[:-1])[:-2], ' ')
                    outline.rstrip()
                    if add_angles:
                        angles = `simuPars['solarZenithAngle']`+" "+\
                                 `simuPars['sensorZenithAngle']`+" "+\
                                 `simuPars['solarSensorAzimuth']`
                        outline += " "+angles
                    outline += "\n"
                    tf.write(outline)
    if red_index!=0 and nir_index!=0:
        # we need to add 1 to the indices since the file already contains the
        #variable in the first column
        addVI(trainingFile, red_index+1, nir_index+1)
```

The utility function to add the NDVI and the RVI to a reflectance ASCII file is like this:

```
def addVI(reflectances_file, red_index, nir_index):
    rff = open(reflectances_file)
    allfields = rff.readlines()
    rff.close()
    with open(reflectances_file, 'w') as rf:
        for l in allfields:
            rfls = string.split(l)
            if len(rfls)>red_index:
                outline = string.join(string.split(l))
                red = float(rfls[red_index-1])
                pir = float(rfls[nir_index-1])
                epsilon = 0.001
                ndvi = (pir-red)/(pir+red+epsilon)
                rvi = pir/(red+epsilon)
                outline += " "+str(ndvi)+" "+str(rvi)+"\n"
                rf.write(outline)
```

The learning the regression model uses the application wrapper:

```
def learnBVModel(trainingFile, outputFile, regressionType, normalizationFile, bestof=1):
    app = otb.Registry.CreateApplication("InverseModelLearning")
    app.SetParameterString("training", trainingFile)
    app.SetParameterString("out", outputFile)
    app.SetParameterString("errest", outputFile+"_errest")
    app.SetParameterString("regression", regressionType)
    app.SetParameterString("normalization", normalizationFile)
    app.SetParameterInt("bestof", bestof)
    app.ExecuteAndWriteOutput()
```

The LAI retrieval from ASCII files for the validation can be performed with the following function:

```
def invertBV(reflectanceFile, modelFile, normalizationFile, outputFile, \
             removeFaparFcover=False, red_index=0, nir_index=0):
    if removeFaparFcover:
        #the reflectance file contains also the simulations of fapar and fcover
        rff = open(reflectanceFile)
        allfields = rff.readlines()
        rff.close()
        with open(reflectanceFile, 'w') as rf:
            for l in allfields:
                outline = string.join(string.split(l)[:-2])+"\n"
                rf.write(outline)

```

```
    if red_index!=0 and nir_index!=0:
        addVI(reflectanceFile, red_index, nir_index)


    app = otb.Registry.CreateApplication("BVInversion")
    app.SetParameterString("reflectances", reflectanceFile)
    app.SetParameterString("model", modelFile)
    app.SetParameterString("normalization", normalizationFile)
    app.SetParameterString("out", outputFile)
    app.ExecuteAndWriteOutput()
```

The use of the `BVImageInversion` application for images would be identical, but the removal of FAPAR and FCover and the addition of the vegetation indices should be implemented using the `BandMath` application which is standard in the ORFEO Toolbox distribution.

2.1.4 No data handling

Pixels which are not flagged as "land" in the L2A products are handled as follows:

- Mono-date LAI: pixels are not processed and they are flagged as `NO_DATA` in the output.
- Multi-date LAI based on "n" last acquisitions: only the valid pixels are used in the computation. If not enough valid dates are available in the estimation window, the mono-date LAI for the current date is used as output. If there was no valid mono-date LAI estimation for the current date, the pixel is marked as `NO_DATA`.
- For the LAI re-processed at the end of the season: only valid pixels are used for the double logistic fitting, the other values being ignored. If there are less than 4 valid dates, the fitting is not performed and the pixels are marked as `NO_DATA`.

2.2 Phenological NDVI metrics

2.2.1 Overview of the subsystem

Figure 3 illustrates the processing chain for the production of the temporal NDVI metrics.

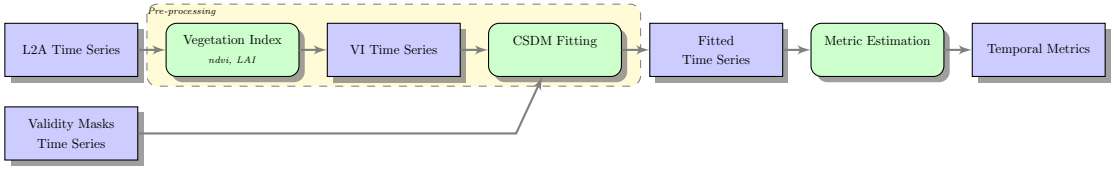

Figure 3: Block diagram for the estimation of the temporal NDVI metrics.

2.2.2 NDVI profile computation

The first step consists in computing the NDVI for each pixel of each date of the L2A image time series. For simplicity, no validity mask is used here. The NDVI computation will use the B8 band (not the B8a). Algorithm

1 describes de procedure.

Algorithm 1: NDVI time series computation

Data: *tocr*
Result: *ndvi_ts*
begin
  $ndvi\_ts \leftarrow \emptyset$;
  for $image \in tocr$ **do**
    $ndvi\_image \leftarrow \emptyset$;
    for $pixel \in image$ **do**
      $ndvi\_image[positon(pixel)] \leftarrow \frac{pixel[NIR]-pixel[R]}{pixel[NIR]+pixel[R]}$;
    end
    $ndvi\_ts \leftarrow ndvi\_ts + ndvi\_image$;
  end
end

2.2.3 Double logistic fitting

The second step consists in fitting a double logistic function (CSDM) to each pixel of the NDVI time series. The validity mask series (clouds, shadows, staurations, etc.) is used to discard the invalid NDVI values. Algorithm 2 describes de procedure.

Algorithm 2: Double logistic fitting

Data: *tocr*, *mask*
Result: *csdm_ts*
begin
  $csdm\_ts \leftarrow \emptyset$;
  for $pixel \in tocr, mask\_pixel \in mask$ **do**
    $v\_pixel \leftarrow remove\_invalid\_dates(pixel, mask\_pixel)$;
    $out\_pixel \leftarrow csdm\_fit(v\_pixel)$;
    $csdm\_ts[positon(pixel)] \leftarrow out\_pixel$;
  end
end

This is done using the `SigmoFitting` application for which the `DoExecute()` method is implemented as follows:

```
1  void DoExecute()
2  {
3    // prepare the vector of dates
4    auto dates = pheno::parse_date_file(this->GetParameterString("dates"));
5    // pipeline
6    FloatVectorImageType::Pointer inputImage = this->GetParameterImage("in");
7    FloatVectorImageType::Pointer maskImage;
8    bool use_mask = true;
9    if(IsParameterEnabled("mask"))
10     maskImage = this->GetParameterImage("mask");
11   else
12     {
13     maskImage = inputImage;
14     use_mask = false;
15     otbAppLogINFO("No mask will be used.\n");
16     }
17   inputImage->UpdateOutputInformation();
18   maskImage->UpdateOutputInformation();
19   bool fit_mode = true;
20   unsigned int nb_out_bands = inputImage->GetNumberOfComponentsPerPixel();
21   if(IsParameterEnabled("mode") && GetParameterString("mode") == "params")
22     {
23     fit_mode = false;
24     nb_out_bands = 12;
```

```
    otbAppLogINFO("Parameter estimation mode.\n");
    }
  using FunctorType =
    pheno::TwoCycleSigmoFittingFunctor<FloatVectorImageType::PixelType>;
  filter = FilterType::New();
  filter->SetInput(0, inputImage);
  filter->SetInput(1, maskImage);
  filter->GetFunctor().SetDates(dates);
  filter->GetFunctor().SetUseMask(use_mask);
  filter->GetFunctor().SetReturnFit(fit_mode);
  filter->SetNumberOfOutputBands(nb_out_bands);
  filter->UpdateOutputInformation();
  SetParameterOutputImage("out", filter->GetOutput());
}
```

The core of the processing (the `csdm_fit` procedure of algorithm 2), is implemented in the `pheno::TwoCycleSigmoFittingFunctor` class. The following type definitions are used:

```
using MinMaxType = std::pair<double, double>;
using CoefficientType = VectorType;
using ApproximationErrorType = double;
using ApproximationResultType = std::tuple<CoefficientType, MinMaxType,
                                           VectorType, VectorType,
                                           ApproximationErrorType>;
```

The interface of the `pheno::TwoCycleSigmoFittingFunctor` is defined as follows:

```
template <typename PixelType>
class TwoCycleSigmoFittingFunctor
{
protected:
  std::vector<tm> dates;
  VectorType dv;
  bool return_fit;
  bool fit_only_invalid;
  bool use_mask;

public:
  struct DifferentSizes {};
  TwoCycleSigmoFittingFunctor() : return_fit{true}, fit_only_invalid{true},
                                  use_mask{true} {};

  void SetDates(const std::vector<tm>& d) {
    // ....
  }
  // ....
  PixelType operator()(PixelType pix, PixelType mask)
  {
    // ....
  }
// ....

};
```

The core of the processing is the `operator()` of the functor:

```
  PixelType operator()(PixelType pix, PixelType mask)
  {
    auto nbDates = pix.GetSize();
    auto tmp_mask = mask;
    if(!use_mask)
      tmp_mask.Fill(0);
    if(dates.size()!=nbDates) throw DifferentSizes{};
    PixelType tmpix{nbDates};
    tmpix.Fill(typename PixelType::ValueType{0});
    // If the mask says all dates are valid, keep the original value
    if(tmp_mask == tmpix && fit_only_invalid && use_mask) return pix;
    VectorType vec(nbDates);
    VectorType mv(nbDates);
    for(auto i=0; i<nbDates; i++)
      {
```

```
      vec[i] = pix[i];
      mv[i] = tmp_mask[i];
      }
    // A date is valid if it is not NaN and the mask value == 0.
    auto pred = [=](int e) { return !(std::isnan(vec[e])) &&
                             (mv[e]==(typename PixelType::ValueType{0})); };
    auto f_profiles = filter_profile(vec, dates, pred);
    decltype(vec) profile=f_profiles.first;
    decltype(vec) t=f_profiles.second;
    // If there are not enough valid dates, keep the original value
    if(profile.size() < 4)
      {
      return pix;
      }
    auto approx = normalized_sigmoid::TwoCycleApproximation(profile, t);
    auto princ_cycle = std::get<1>(approx);
    auto x_hat = std::get<0>(princ_cycle);
    auto min_max = std::get<1>(princ_cycle);
    auto A_hat = min_max.second - min_max.first;
    auto B_hat = min_max.first;

    double dgx0, t0, t1, t2, t3, dgx2;
    std::tie(dgx0, t0, t1, t2, t3, dgx2) =
      pheno::normalized_sigmoid::pheno_metrics<double>(x_hat, A_hat, B_hat);

    // The metrics have to fulfill some constraints in order to be
    //    considered as valid: t0<x0<t1<t2<t3 and (t3 - t0) < 365
    if((t0 < x_hat[0]) && (x_hat[0] < t1) && (t1 < t2) &&
       (t2 < t3) && ((t3 - t0) < 365))
      {
      result[0] = x_hat[0];
      result[1] = t0;
      result[2] = (t2-t1);
      result[3] = t3;
      }
    return result;
      }
  }
```

The approximation is done in the `normalized_sigmoid::TwoCycleApproximation(profile, t)` call:

```
template <ContainerC V, DateContainerC W>
std::tuple<VectorType, ApproximationResultType, ApproximationResultType>
TwoCycleApproximation(const V& profile, const W& t)
{
  auto pca = PrincipalCycleApproximation(profile, t);
  // Approximate the residuals
  auto residual_approx = Approximation(std::get<3>(pca), t);
  // The approximated profile is the sum of the approximations
  auto yHat = std::get<2>(pca)+std::get<2>(residual_approx);
  // Return {x, minmax, yHat, residuals, err};
  return std::make_tuple(yHat, pca, residual_approx);
}
```

The `PrincipalCycleApproximation` can be implemented as follows:

```
template <ContainerC V, DateContainerC W>
ApproximationResultType PrincipalCycleApproximation(const V& profile,
                                                    const W& t)
{
  // Profile approximation
  auto approx_result = Approximation(profile, t);
  auto mm = std::get<1>(approx_result);
  auto residuals = std::get<3>(approx_result)+mm.first;

  // Residual approximation
  approx_result = Approximation(residuals, t);


  // Subtract the residual approximation from the original profile
  auto yHat = std::get<2>(approx_result);
  residuals = profile-yHat;

```

```
  // Approx the original minus the residuals
  approx_result = Approximation(residuals, t);

  // Get the data to be returned
  auto x = std::get<0>(approx_result);
  auto minmax = std::get<1>(approx_result);
  yHat = std::get<2>(approx_result);
  residuals = profile-yHat;
  auto err = std::get<4>(approx_result);

  return ApproximationResultType{x, minmax, yHat, residuals, err};
}
```

And the approximation is obtained by using the `Approximation` function:

```
template <ContainerC V>
ApproximationResultType Approximation(const V& profile, const V& t)
{
  auto minmax = std::minmax_element(std::begin(profile), std::end(profile));
  auto t_max = std::begin(t);
  std::advance(t_max,std::distance(std::begin(profile),minmax.second));
  auto vmax = *(minmax.second);
  auto vmin = *(minmax.first);
  auto prof = (profile-vmin)/(vmax-vmin);
  auto x(guesstimator(prof, t));
  auto fprofile = gaussianWeighting(prof, t, *t_max,75.0);
  auto phFs(F<V>);
  ParameterCostFunction fs{4, t.size(), fprofile, t, phFs};
  auto err(optimize(x, fs));
  auto yHat(phFs(t,x));
  auto residuals = prof - yHat;
  yHat = yHat*(vmax-vmin)+vmin;
  residuals = residuals*(vmax-vmin);
  auto mm = std::make_pair(vmin, vmax);
  return ApproximationResultType{x, mm, yHat, residuals, err};
}
```

The `guesstimator()` function initializes the values of the parameters:

```
template <ContainerC V>
inline
V guesstimator(V p, V t)
{
  // Find the max position
  auto pmax = std::max_element(std::begin(p), std::end(p)) - std::begin(p);
  V x0(4);
  x0[0] = t[pmax] - 25;
  x0[1] = 10.0;
  x0[2] = t[pmax] + 25;
  x0[3] = 10.0;
  return x0;
}
```

The `gaussianWeighting()` function reduces the values far from the principal peak of the profile:

```
template <ContainerC V>
inline
V gaussianWeighting(const V& p, const V& t, double m, double s)
{
  auto fp = p;
  for(auto i=0; i<t.size(); i++)
    {
    fp[i] *= gaussianFunction(t[i],m,s);
    }

  return fp;
}
```

And the `gaussianFunction()` is just the well known Gaussian:

```
double gaussianFunction(double x, double m, double s)
{
  return exp(-vnl_math_sqr((x-m)/(s)/2.0));
}
```

The parameters are optimized using a Levenberg Marquardt optimizer available in the Orfeo Toolbox via the VNL libraries:

```
double optimize(VectorType& x, ParameterCostFunction f)
{
  vnl_levenberg_marquardt levmarq(f);
  levmarq.set_f_tolerance(1e-10);
  levmarq.set_x_tolerance(1e-100);
  levmarq.set_g_tolerance(1e-100);
  levmarq.minimize(x);
  return levmarq.get_end_error();
}
```

The `ParameterCostFunction` is

```
class ParameterCostFunction : public vnl_least_squares_function
{
public:
  ParameterCostFunction(unsigned int nbPars, unsigned int nbD,
                        const VectorType& yy, const VectorType& tt,
                        FunctionType func) :
    vnl_least_squares_function(nbPars, nbD, no_gradient), nbDates(nbD), y(yy),
    t(tt), phenoFunction(std::move(func)) {}

  inline
  void f(const VectorType& x, VectorType& fx) override
  {
    auto yy = phenoFunction(t, x);
    for(auto i=0; i<nbDates; ++i)
      fx[i] = (y[i] - yy[i]);
  }

private:
  VectorType y;
  VectorType t;
  unsigned int nbDates;
  FunctionType phenoFunction;
};
```

And in our case, the function to be optimized is

```
namespace normalized_sigmoid{
template <typename T, ContainerC V>
inline
T double_sigmoid(T t, const V& x)
{
  return 1.0/(1.0+exp((x[0]-t)/x[1]))-1.0/(1.0+exp((x[2]-t)/x[3]));
}

template <ContainerC V>
inline
V F(const V& t, const V& x)
{
  auto tsize(t.size());
  V y(tsize);
  for(auto i=0; i<tsize; ++i)
    y[i] = double_sigmoid(t[i], x);
  return y;
}
}
```

2.2.4 Metric estimations

The following phenological parameters were selected as meaningful to describe the vegetation status: emergence date, date of the maximum growth, length of the maturity plateau and senescence date.

The logistic function has the form:

$$f(x) = \frac{1}{1 + e^{\frac{x_0 - x}{x_1}}} \tag{1}$$

The double logistic is:

$$g(x) = A\left(f_1(x) - f_2(x)\right) + B = A\left(\frac{1}{1 + e^{\frac{x_0 - x}{x_1}}} - \frac{1}{1 + e^{\frac{x_2 - x}{x_3}}}\right) + B, \tag{2}$$

where $A + B$ is the maximum value and B is the minimum.
Since

$$\frac{df(x)}{dx} = \frac{e^{\frac{x_0 - x}{x_1}}}{x_1\left(1 + e^{\frac{x_0 - x}{x_1}}\right)^2} \tag{3}$$

we have

$$g'(x) = \frac{dg(x)}{dx} = A\left(\frac{e^{\frac{x_0 - x}{x_1}}}{x_1\left(1 + e^{\frac{x_0 - x}{x_1}}\right)^2} - \frac{e^{\frac{x_2 - x}{x_3}}}{x_3\left(1 + e^{\frac{x_2 - x}{x_3}}\right)^2}\right) \tag{4}$$

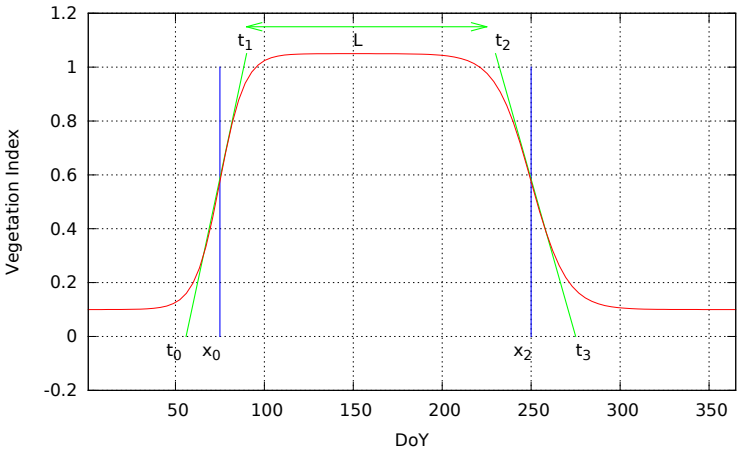

Figure 4: Double logistic function and associated parameters

2.2.4.1 Date of the maximum positive gradient By definition, this is x_0.

2.2.4.2 Starting date The date for which the straight line with the slope of x_0 intercepts the horizontal axis. If

$$g'(x_0) = m \tag{5}$$

this line's equation is

$$y = mx + b \tag{6}$$

and verifies that

$$\begin{aligned} 0 &= mt_0 + b \\ g(x_0) &= mx_0 + b \end{aligned} \tag{7}$$

which gives

$$y = g'(x_0)x + (g(x_0) - g'(x_0)x_0) \tag{8}$$

and therefore,

$$t_0 = \frac{mx_0 - g(x_0)}{m} = x_0 - \frac{g(x_0)}{g'(x_0)} \tag{9}$$

2.2.4.3 Length of the plateau We define t_1 as the date for which the previous straight line reaches the maximum value.

$$t_1 = \frac{A + B - (g(x_0) - g'(x_0)x_0)}{g'(x_0)} \tag{10}$$

Similarly, we can use the straight line associated to the descending slope:

$$y = g'(x_2)x + (g(x_2) - g'(x_2)x_2) \tag{11}$$

and define

$$t_2 = \frac{A + B - (g(x_2) - g'(x_2)x_2)}{g'(x_2)} \tag{12}$$

And the length of the plateau is:

$$L = t_2 - t_1 \tag{13}$$

2.2.4.4 Senescence date In a similar way than for t_0, we obtain:

$$t_3 = x_2 - \frac{g(x_2)}{g'(x_2)} \tag{14}$$

2.2.4.5 Metrics estimation implementation Using the definitions above, algorithm 3 describes the metric estimation loop.

Algorithm 3: NDVI time metrics computation

Data: *csdm_ts*
Result: *metrics_image*
begin
  metrics_image $\leftarrow \emptyset$;
  for *pixel* $\in$ *csdm_ts* **do**
    out_pixel $\leftarrow$ *pheno_metrics*(*pixel*);
    metrics_image[*positon*(*pixel*)] $\leftarrow$ *out_pixel*;
  end
end

This alorithm is implemented in terms of several functions. The derivative of the double logistic is:

```
template <typename T>
inline
T diff_sigmoid(T t, T x0, T x1)
{
  auto b = exp((x0-t)/x1);
  return 1.0/(x1*(1+1/b)*(1+b));
}

template <typename T, ContainerC V>
inline
T diff_double_sigmoid(T t, const V& x)
{
  return diff_sigmoid(t, x[0], x[1])-diff_sigmoid(t, x[2], x[3]);
}
```

And, finally the metrics estimation is just a straighforward FORmula TRANslation, but in C++:

```
template <typename T, ContainerC V>
inline
std::tuple<T, T, T, T, T, T> pheno_metrics(const V& x, T maxvalue=1.0,
                                           T minvalue=0.0)
{
  auto A = maxvalue-minvalue;
  auto B = minvalue;
  auto gx0 = A*double_sigmoid(x[0], x)+B;
  auto dgx0 = A*diff_double_sigmoid(x[0], x);
  auto gx2 = A*double_sigmoid(x[2], x)+B;
  auto dgx2 = A*diff_double_sigmoid(x[2], x);

  auto t0 = x[0] - gx0/dgx0;
  auto t1 = (A+B-(gx0-dgx0*x[0]))/dgx0;
  auto t2 = (A+B-(gx2-dgx2*x[2]))/dgx2;
  auto t3 = x[2] - gx2/dgx2;

  return std::make_tuple(dgx0, t0, t1, t2, t3, dgx2);
}
```

2.2.4.6 No data handling and metric checking

The double logistic fitting is applied on all the pixels for which the number of valid dates is equal or greater than 4. The invalid dates are ignored and ony the valid dates are used for the fitting.

For some NDVI time profiles with unexpected behaviours (absence of vegetation, very atypical shapes), the estimation of the metrics could yield unrealistic values.

A post-processing checking will be implemented in order to detect wrong estimations. This post-processing consists in checking that both the following conditions are true:

$$t0 < x0 < t1 < t2 < t3;$$

$$t3 - t0 < 365.$$

The values of the pixels not fulfilling these conditions will be set to `NO_DATA`.

2.2.5 Near-real-time emergence date estimation

After the literature review and the benchmarking phase, no satisfactory approach for the estimation of the emergence date in near-real-time was found. Furthermore, no suitable data for the validation of a new algorithm was available for the SPOT4 (Take5) data set. The SPOT5 (Take5) data sets may make in situ data available for the prototyping and validation of a simple thresholding approach.

The final algorithm will be described in an updated version of the ATBD.

Since the literature indicates that thresholding approaches have to be crop dependent (see section 2.5.3 of the DJF about the vegetation status product), this approach will not be reliable enough to be included in the system. It will therefore be provided as an external module.

2.3 Quality flags

- For the mono-date LAI maps:
  - The flag that gives the status (no_{data}, cloud, snow, water, land, cloud shadow and saturation).
  - The uncertainty, which is the output of the error estimation.
- For the multi-date LAI based on "n" last acquisitions:
  - The number of dates (i.e. L2A products) which are associated with the "land" status during the search window; knowing that the multi-date LAI will not be computed if this number is lower than 2). There is a flag for each date in the time profile. Therefore, this value will be between 3 and n if several dates are used. It will be equal to 1 if less than 3 valid dates were availble and the LAI

mono-date existed for the current date, and 0 otherwise. In this last case, the LAI value will be set to `NO_DATA`.

- For the LAI re-processed at the end of the season and for the NDVI metrics:
  - The number of dates (i.e. L2A products) which are associated with the "land" status during the season; knowing that the re-processed LAI and the NDVI metrics will not be computed if this number is lower than 4).
- For the NDVI metrics:
  - The number of dates (i.e. L2A products) which are associated with the "land" status during the interval.

3 Appendix

3.1 The OTB-BV library

This section describes the C++ library which implements the low level details of the LAI retrieval. This library makes extensive use of the ORFEO Toolbox.

3.1.1 Type definitions

A definition of a set of types is given in `otbBVTypes.h`:

```
namespace otb
{
enum class IVNames {MLAI, ALA, CrownCover, HsD, N, Cab, Car, Cdm, CwRel, Cbp,
                    Bs, IVNamesEnd};
enum AcquisitionParameters {TTS, TTO, PSI, TTS_FAPAR, AcquisitionParametersEnd};

using AcquisitionParsType = std::map< AcquisitionParameters, double >;
using PrecisionType = double;
using BVType = std::map< IVNames, PrecisionType >;

using NormalizationVectorType =
  std::vector<std::pair<PrecisionType, PrecisionType>>;
}
```

Solar irradiance values for the FAPAR computation in the simulations (not needed for LAI) are given in \ `otbSolarIrradianceFAPAR.h` :

```
namespace otb{
static
std::vector<std::pair<PrecisionType, PrecisionType>> solar_irradiance_fapar = {
  {0.4000, 1614.0900},
  {0.4025, 1631.0900},
  {0.4050, 1648.0900},
  // etc.
  {0.7000, 1408.2400}
};
}
#endif
```

3.1.2 Utilities

A utility function for counting columns in an ASCII file is given in `otbBVUtil.h`. It is used to automatically get the number of predictor variables in the non-linear regression procedures.

```
namespace otb
{
  unsigned short int countColumns(std::string fileName)
  {
    std::ifstream ifile(fileName.c_str());
    std::string line;
    auto nbSpaces = 0;
    if (ifile.is_open())
      {
      getline(ifile,line);
      ifile.close();
      boost::trim(line);
      auto found = line.find(' ');
      while(found!=std::string::npos)
        {
        ++nbSpaces;
        found = line.find(' ', found+1);
        }
      return nbSpaces+1;
      }
    else
      {
      itkGenericExceptionMacro(<< "Could not open file " << fileName);
      }

  }
}
```

3.1.3 Simulation

The file `otbProSailSimulatorFunctor.h` contains the definition of a functor which performs the simulation of the reflectances for a pixel. The simulation is delegated to the PROSPECT and Sail implementations available in the ORFEO Toolbox.

The interface of the class is as follows:

```
namespace otb
{
namespace Functor
{

template <class TSatRSR, unsigned int SimNbBands = 2000>
class ProSailSimulator
{
public:
  /** Standard class typedefs */
  typedef TSatRSR SatRSRType;
  typedef typename SatRSRType::Pointer SatRSRPointerType;
  typedef typename otb::ProspectModel ProspectType;
  typedef typename otb::LeafParameters LeafParametersType;
  typedef typename LeafParametersType::Pointer LeafParametersPointerType;
  typedef typename otb::SailModel SailType;

  typedef typename SatRSRType::PrecisionType PrecisionType;
  typedef std::pair<PrecisionType,PrecisionType> PairType;
  typedef typename std::vector<PairType> VectorPairType;
  typedef otb::SpectralResponse< PrecisionType, PrecisionType>  ResponseType;
  typedef otb::ReduceSpectralResponse < ResponseType,SatRSRType>
                                                    ReduceResponseType;
  typedef typename std::vector<PrecisionType> OutputType;

  inline
  OutputType operator ()()
    protected:

     double ComputeFAPAR(SailType::SpectralResponseType* absorptance);
  /** Satellite Relative spectral response*/
  SatRSRPointerType m_SatRSR;
  LeafParametersPointerType m_LP;
  double m_LAI; //leaf area index
  double m_Angl; //average leaf angle
  double m_PSoil; //soil coefficient
```

```
  double m_Skyl; //diffuse/direct radiation
  double m_HSpot; //hot spot
  double m_TTS; //solar zenith angle
  double m_TTS_FAPAR; //solar zenith angle for fapar computation
  double m_TTO; //observer zenith angle
  double m_PSI; //azimuth
  BVType m_BV;
};

}
}
```

The processing is done in the `operator()` member which is implemented as follows:

```
  inline
  OutputType operator ()()
  {
    OutputType pix;
    for(auto i=0;i<m_SatRSR->GetNbBands();i++)
      pix.push_back(0.0);
    // ....
    auto prospect = ProspectType::New();
    // ....
    auto refl = prospect->GetReflectance()->GetResponse();
    auto trans = prospect->GetTransmittance()->GetResponse();
    auto sail = SailType::New();
    // ....
    auto sailSim = sail->GetViewingReflectance()->GetResponse();
    auto fCover = sail->GetFCoverView();
    auto sail_fapar = SailType::New();
    // ....
    auto fAPAR = this->ComputeFAPAR(sail_fapar->GetViewingAbsorptance());
    VectorPairType hxSpectrum;
    for(auto i=0;i<SimNbBands;i++)
      {
      PairType resp;
      resp.first = static_cast<PrecisionType>((400.0+i)/1000);
      resp.second = sailSim[i].second;
      hxSpectrum.push_back(resp);
      }
    auto aResponse = ResponseType::New();
    aResponse->SetResponse( hxSpectrum );
    auto  reduceResponse = ReduceResponseType::New();
    reduceResponse->SetInputSatRSR(m_SatRSR);
    reduceResponse->SetInputSpectralResponse( aResponse );
    reduceResponse->SetReflectanceMode(true);
    reduceResponse->CalculateResponse();
    for(auto i=0;i<m_SatRSR->GetNbBands();i++)
      pix[i] = (*reduceResponse)(i);
    pix.push_back(fCover);
    pix.push_back(fAPAR);
    return pix;
  }
```

The commented (`// ....`) sections above correspond to the parameter settings for PROSPECT and Sail. For the leaf parameters (PROSPECT) we have:

```
    auto m_LP = LeafParametersType::New();
    m_LP->SetCab(m_BV[IVNames::Cab]);
    m_LP->SetCar(m_BV[IVNames::Car]);
    m_LP->SetCBrown(m_BV[IVNames::Cbp]);
    double Cw = m_BV[IVNames::Cdm]/(1.-m_BV[IVNames::CwRel]);
    if(Cw<0) Cw = 0.0;
    m_LP->SetCw(Cw);
    m_LP->SetCm(m_BV[IVNames::Cdm]);
    m_LP->SetN(m_BV[IVNames::N]);
```

And the Sail parameters are set as follows:

```
    m_LAI = m_BV[IVNames::MLAI];
    m_Angl = m_BV[IVNames::ALA];
    m_PSoil = m_BV[IVNames::Bs];
    m_Skyl = 0.3;
    m_HSpot = m_BV[IVNames::HsD];
    auto sail = SailType::New();
    sail->SetLAI(m_LAI);
    sail->SetAngl(m_Angl);
    sail->SetPSoil(m_PSoil);
    sail->SetSkyl(m_Skyl);
    sail->SetHSpot(m_HSpot);
    sail->SetTTS(m_TTS);
    sail->SetTTO(m_TTO);
    sail->SetPSI(m_PSI);
    sail->SetReflectance(prospect->GetReflectance());
    sail->SetTransmittance(prospect->GetTransmittance());
    sail->Update();
```

3.1.4 Regression

The non-linear regression is implemented using Neural Networks or Support Vector Machines. In both cases, the OpenCV implementation available through the ORFEO Toolbox is used. However, the ORFEO Toolbox wrapper for the `NeuralNetworkMachineLearningModel` does not allow to perform regression (only classification is provided), and therefore, a `NeuralNetworkRegressionMachineLearningModel` has been implemented.

It boils down to inheriting and overloading the `Predict` method in order to do regression:

```
namespace otb
{
template <class TInputValue, class TTargetValue>
class ITK_EXPORT NeuralNetworkRegressionMachineLearningModel
  : public NeuralNetworkMachineLearningModel <TInputValue, TTargetValue>
{
// ....
};
}
```

```
template<class TInputValue, class TOutputValue>
typename
NeuralNetworkRegressionMachineLearningModel<TInputValue,
                                            TOutputValue>::TargetSampleType
NeuralNetworkRegressionMachineLearningModel<TInputValue,
                                            TOutputValue>::Predict(
                                              const
                                              InputSampleType & input) const
{
  //convert listsample to Mat
  cv::Mat sample;
  otb::SampleToMat<InputSampleType>(input, sample);
  cv::Mat response; //(1, 1, CV_32FC1);
  m_ANNModel->predict(sample, response);
  TargetSampleType target;
  target[0] = response.at<float> (0, 0);
  return target;
}
```

3.1.5 Reprocessing

Reprocessing functions working on time profiles are defined in\ `otbProfileReprocessing.h`.

3.1.5.1 On-line reprocessing The on-line reprocessing uses a local window defined by its backward radius `bwd_radius` and its forward radius `fwd_radius`. For example, if $(bwd\_radius, fwd\_radius) = (1, 1)$ the window contains 3 dates and is centered on the middle one.

The `smooth_time_series_local_window_with_error` takes as input a date vector `dts`, the vector with the measurement values (i.e. LAI) `ts`, and the vector of the estimated errors `ets`. For each reprocessed date, the new value is computed using a weighted linear combination of the values inside the window.

```
std::pair<VectorType, VectorType>
smooth_time_series_local_window_with_error(VectorType dts,
                                           VectorType ts,
                                           VectorType ets,
                                           size_t bwd_radius = 1,
                                           size_t fwd_radius = 1)
{

  /**
        ------------------------------------
        |                   |             |
      win_first           current       win_last
  */
  assert(ts.size()==ets.size() && ts.size()==dts.size());
  auto result = ts;
  auto result_flag = ts;
  auto ot = result.begin();
  auto otf = result_flag.begin();
  auto eit = ets.begin();
  auto last = ts.end();
  auto win_first = ts.begin();
  auto win_last = ts.begin();
  auto e_win_first = ets.begin();
  auto e_win_last = ets.begin();
  auto dti = dts.begin();
  auto d_win_first = dts.begin();
  auto d_win_last = dts.begin();
  *otf = not_processed_value;
  //advance iterators
  std::advance(ot, bwd_radius);
  std::advance(otf, bwd_radius);
  std::advance(eit, bwd_radius);
  std::advance(dti, bwd_radius);
  std::advance(win_last, bwd_radius+fwd_radius);
  std::advance(e_win_last, bwd_radius+fwd_radius);
  std::advance(d_win_last, bwd_radius+fwd_radius);
  while(win_last!=last)
    {
    auto current_d = d_win_first;
    auto current_e = e_win_first;
    auto current_v = win_first;
    auto past_it = d_win_last; ++past_it;

    PrecisionType sum_weights{0.0};
    PrecisionType weighted_value{0.0};
    while(current_d != past_it)
      {
      auto cw = compute_weight(fabs(*current_d-*dti),fabs(*current_e));
      sum_weights += cw;
      weighted_value += (*current_v)*cw;
      ++current_d;
      ++current_e;
      ++current_v;
      }
    *ot = weighted_value/sum_weights;
    *otf = processed_value;
    ++win_first;
    ++win_last;
    ++e_win_first;
    ++e_win_last;
    ++d_win_first;
    ++d_win_last;
    ++ot;
    ++otf;
    ++eit;
    ++dti;
    }
  *otf = not_processed_value;
  return std::make_pair(result,result_flag);
}
```

The compute weight function is:

```
template <typename T>
inline
T compute_weight(T delta, T err)
{
  T one{1};
  return (one/(one+delta)+one/(one+err));
}
```

3.1.5.2 Fit CSDM The double logistic fitting is performed using functions provided by the `phenotb` library which is described in section 2.2. These functions are used by the `fit_csdm` function below, which takes as input a date vector `dts`, the vector with the measurement values (i.e. LAI) `ts`. A third parameter `ets` corresponding to the vector of the estimated errors is not used in the current version of the function.

The function returns a `std::pair` of vectors containing the fitted profile and a vector of flags for each date inidicating whether or not each date has been processed.

```
std::pair<VectorType, VectorType>
fit_csdm(VectorType dts, VectorType ts, VectorType ets)
{
  assert(ts.size()==ets.size() && ts.size()==dts.size());
  auto result = ts;
  auto result_flag = ts;
  // std::vector to vml_vector
  pheno::VectorType profile_vec(ts.size());
  pheno::VectorType date_vec(dts.size());

  for(auto i=0; i<ts.size(); i++)
    {
    Profile_vec[i] = ts[i];
    date_vec[i] = dts[i];
    }

  // fit
  auto approximation_result =
    pheno::normalized_sigmoid::TwoCycleApproximation(profile_vec, date_vec);
  auto princ_cycle = std::get<1>(approximation_result);
  auto x_hat = std::get<0>(princ_cycle);
  auto min_max = std::get<1>(princ_cycle);
  auto A_hat = min_max.second - min_max.first;
  auto B_hat = min_max.first;
  auto p = pheno::normalized_sigmoid::F(date_vec, x_hat);
  //fill the result vectors
  for(auto i=0; i<ts.size(); i++)
    {
    result[i] = p[i]*A_hat+B_hat;
    result_flag[i] = processed_value;
    }

  return std::make_pair(result,result_flag);
}
```