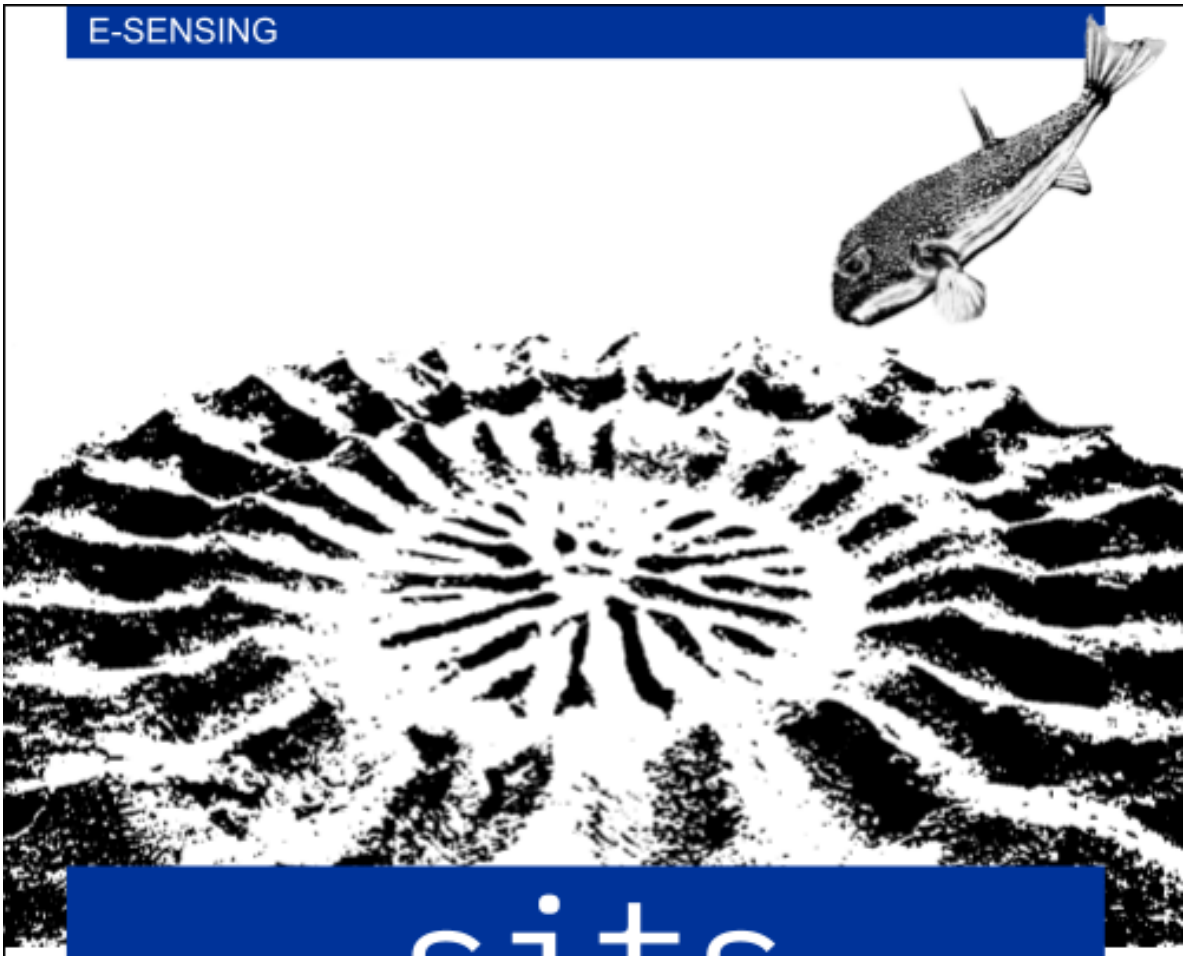
sits: Satellite Image Time Series Analysis on Earth Observation Data Cubes

Gilberto Camara Rolf Simoes Felipe Souza Charlotte Peletier
Alber Sanchez Pedro R. Andrade Karine Ferreira
Gilberto Queiroz

2022-07-09

E-SENSING



sits

Satellite Image Time Series Analysis on Earth Observation Data Cubes

Gilberto Camara, Rolf Simoes, Felipe Souza,
Charlotte Peletier, Alber Sanchez,
Pedro Ribeiro de Andrade,
Karine Ferreira, Gilberto Queiroz

Contents

List of Figures

List of Tables

Preface

Using time series derived from big Earth Observation data sets is one of the leading research trends in Land Use Science and Remote Sensing. One of the more promising uses of satellite time series is its application to classify land use and land cover. Information on land is critical for sustainable development because our growing demand for natural resources is causing significant environmental impacts. As stated by [1], *"time series analysis is expanding the kinds of land surface change that can be monitored using remote sensing. More subtle changes in ecosystem health and condition and related to land use dynamics are being monitored. The result is a paradigm shift away from change detection, typically using two points in time, to monitoring, or an attempt to track change continuously in time"*.

This book presents `sits`, an open-source R package for land use and land cover classification using big Earth observation data. Users can build regular data cubes from collections in AWS, Microsoft Planetary Computer, Brazil Data Cube, and Digital Earth Africa. The software includes functions for quality assessment of training samples using self-organized maps. It provides machine learning and deep learning algorithms for classification of big Earth observation data cubes. Post-processing methods includes Bayesian smoothing and uncertainty assessment. Users can apply best practices for estimating area and assessing accuracy of land change. Thus, `sits` is an end-to-end toolkit for land mapping with Earth observation.

Who this book is for

The target audience for **sits** is the new generation of specialists who understand the principles of remote sensing and can write scripts in **R**. Ideally, users should have basic knowledge of data science methods using R. If you want more information on methods used in this book, please look at the following references:

- Garrett Golemund, "Hands-On Programming with R". O'Reilly, 2014. https://rstudio-education.github.io/hopr/.

- Hadley Wickham and Gareth Golemund, "R for Data Science". O'Reilly, 2017. https://r4ds.had.co.nz/.
- Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, "An Introduction to Statistical Learning with Applications in R", Springer, 2013. https://www.statlearning.com/.
- Zhang, A.; Zachary C. Lipton, Mu Li, Alexander J. Smola, "Dive into Deep Learning", 2021. https://d2l.ai/.

How to use this book

This book describes **sits** version 1.1.0. Download and install the package as explained in the Setup section. Start at Chapter 1 to get an overview of the package. Then feel free to browse the chapter for more information on topics you are interested in.

Chapter	Description
Chr 1[1]	Provides an overview of **sits**.
Chr 2[2]	Describes how to work with Earth observation data cubes.
Chr 3[3]	Describes how to access information from time series.
Chr 4[4]	Improving the quality of the samples used in training models
Chr 5[5]	Presents the machine learning techniques available in sits.
Chr 6[6]	Describes how to classify satellite images associated with Earth observation data cubes and smoothing methods to reclassify the pixels based on the machine learning probabilities.
Chr 7[7]	Presents the validation and accuracy measures.
Chr 8[8]	Presents case studies of LUCC classification.
Chr 9[9]	How to develop extensions to **sits**.

Main reference for sits

If you use sits in your work, please cite the following paper:

Rolf Simoes, Gilberto Camara, Gilberto Queiroz, Felipe Souza, Pedro R. Andrade, Lorena Santos, Alexandre Carvalho, and Karine Ferreira. "Satellite Image Time Series Analysis for Big Earth Observation Data". Remote Sensing, 13, p. 2428, 2021. https://doi.org/10.3390/rs13132428.

Setup

The sits package relies on the sf and terra R packages, which in turn require the GDAL and PROJ libraries. Please follow the instructions below for installing sf and terra together with GDAL, provided by Edzer Pebesma.

Support for GDAL and PROJ

Windows and MacOS

Windows and MacOS users are strongly encouraged to install the sf and terra binary packages from CRAN. To install sits from source, please install package Rtools to have access to the compiling environment.

Ubuntu

For Ubuntu versions 20.04 (focal) and 22.04 (jammy), we provide a script for fast installation, based on the work by Dirk Eddelbuettel in r2u[10] project. The script installs binary versions of sits and all its dependencies. It requires R version 4.2.0 or later. To download the script use:

```
wget https://raw.githubusercontent.com/e-sensing/sitsbook/master/utils/
    fast_sits_installation_ubuntu.sh
```

and run the script as sudo:

```
sudo sh fast_sits_installation_ubuntu.sh
```

Alternatively, the installation can be done step-by-step. We recommend using the latest version of the GDAL, GEOS, and PROJ4 libraries. To do so, use the repository ubuntugis-unstable, which should be done as follows:

```
sudo add-apt-repository ppa:ubuntugis/ubuntugis-unstable
sudo apt-get update
sudo apt-get install libudunits2-dev libgdal-dev libgeos-dev libproj-dev
```

[10] https://github.com/eddelbuettel/r2u

If you get an error while adding this PPA repository, it could be because you miss the package `software-properties-common`. When GDAL is running in `docker` containers, please add the security flag `--security-opt seccomp=unconfined` on start.

Debian

To install on Debian, use the rocker geospatial[11] dockerfiles.

Fedora

The following command installs all required dependencies:

```
sudo dnf install gdal-devel proj-devel geos-devel sqlite-devel udunits2-
    devel
```

Installing the `sits` package

`sits` is available on CRAN and should be installed as other R packages.

```
install.packages("sits")
```

The source code repository is on GitHub https://github.com/e-sensing/sits. To install the development version of `sits`, which contains the latest updates but might be unstable, users should install `devtools` if not already available, as then install sits as follows:

```
install.packages("devtools")
devtools::install_github("e-sensing/sits@dev", dependencies = TRUE)
```

To run the examples in the book, please also install the "sitsdata" package.

```
options(download.file.method = "wget")
devtools::install_github("e-sensing/sitsdata")
```

[11] https://github.com/rocker-org/geospatial

Acknowledgements

Funding Sources

The authors acknowledge the funders that supported the development of `sits`:

1. Amazon Fund, established by the Brazil with financial contribution from Norway, through contract 17.2.0536.1. between the Brazilian Development Bank (BNDES) and the Foundation for Science, Technology and Space Applications (FUNCATE), for the establishment of the Brazil Data Cube,

2. Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES) and from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for grants 312151/2014-4 and 140684/2016-6.

3. Sao Paulo Research Foundation (FAPESP) under eScience Program grant 2014/08398-6, for for providing MSc, PhD and post-doc scholarships, equipment, and travel support.

4. International Climate Initiative of the Germany Federal Ministry for the Environment, Nature Conservation, Building and Nuclear Safety (IKI) under grant 17-III-084- Global-A-RESTORE+ ("RESTORE+: Addressing Landscape Restoration on Degraded Land in Indonesia and Brazil").

5. Microsoft Planetary Computer initiative under the GEO-Microsoft Cloud Computer Grants Programme.

Community Contributions

The authors thank Marius Appel, Tim Appelhans, Henrik Bengtsson, Robert Hijmans, Edzer Pebesma, and Ron Wehrens, for their R packages `gdalcubes`, `leafem`, `data.table`, `terra`, `sf`/`stars`, and "kohonen". We are indebted to the RStudio team, including Hadley Wickham for the `tidyverse` and Daniel Falbel for `torch`.

Reproducible papers used in building sits

We thank the authors of these papers for making their code available.

- Marius Appel and Edzer Pebesma, "On-Demand Processing of Data Cubes from Satellite Image Collections with the Gdalcubes Library." Data 4 (3): 1–16, 2020. https://doi.org/10.3390/data4030092

- Hassan Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller, "Deep learning for time series classification: a review". Data Mining and Knowledge Discovery, 33(4): 917–963, 2019. https://doi.org/10.1007/s10618-019-00619-1.

- Edzer Pebesma, "Simple Features for R: Standardized Support for Spatial Vector Data". R Journal, 10(1):2018.

- Charlotte Pelletier, Geoffrey Webb, and Francois Petitjean. "Temporal Convolutional Neural Network for the Classification of Satellite Image Time Series". Remote Sensing 11 (5), 2019. https://doi.org/10.3390/rs11050523

- Ron Wehrens and Kruisselbrink, Johannes. "Flexible Self-Organising Maps in kohonen 3.0". Journal of Statistical Software, 87, 7 (2018). https://doi.org/10.18637/jss.v087.i07.

- Vivien Garnot, Loic Landrieu, Sebastien Giordano, and Nesrine Chehata, "Satellite Image Time Series Classification with Pixel-Set Encoders and Temporal Self-Attention", Conference on Computer Vision and Pattern Recognition, 2020. https://doi.org/10.1109/CVPR42600.2020.01234.

- Vivien Garnot, Loic Landrieu, "Lightweight Temporal Self-Attention for Classifying Satellite Images Time Series", 2020. <arXiv:2007.00586>.

- Maja Schneider, Marco Körner, "[Re] Satellite Image Time Series Classification with Pixel-Set Encoders and Temporal Self-Attention." ReScience C 7 (2), 2021. doi:10.5281/zenodo.4835356.

Publications using sits

This section gathers the publications that have used **sits** to generate their results.

2021

- Lorena Santos, Karine R. Ferreira, Gilberto Camara, Michelle Picoli, Rolf Simoes, "Quality control and class noise reduction of satellite image time series". ISPRS Journal of Photogrammetry and Remote Sensing, vol. 177, pp 75-88, 2021. https://doi.org/10.1016/j.isprsjprs.2021.04.014.

- Lorena Santos, Karine Ferreira, Michelle Picoli, Gilberto Camara, Raul Zurita-Milla and Ellen-Wien Augustijn, "Identifying Spatiotemporal Patterns in Land Use and Cover Samples from Satellite Image Time Series". Remote Sensing, 2021, 13(5), 974; https://doi.org/10.3390/rs13050974.

2020

- Rolf Simoes, Michelle Picoli, Gilberto Camara, Adeline Maciel, Lorena Santos, Pedro Andrade, Alber Sánchez, Karine Ferreira & Alexandre Carvalho. "Land use and cover maps for Mato Grosso State in Brazil from 2001 to 2017". Nature Scientific Data 7, article 34 (2020). DOI: 10.1038/s41597-020-0371-4.

- Michelle Picoli, Ana Rorato, Pedro Leitão, Gilberto Camara, Adeline Maciel, Patrick Hostert, Ieda Sanches, "Impacts of Public and Private Sector Policies on Soybean and Pasture Expansion in Mato Grosso—Brazil from 2001 to 2017". Land, 9(1), 2020. DOI: 10.3390/land9010020.

- Karine Ferreira, Gilberto Queiroz et al., "Earth Observation Data Cubes for Brazil: Requirements, Methodology and Products". Remote Sensing, 12, 4033, 2020.

- Adeline Maciel, Lubia Vinhas, Michelle Picoli and Gilberto Camara, "Identifying Land Use Change Trajectories in Brazil's Agricultural Frontier". Land, 9, 506, 2020. DOI: 10.3390/land9120506. DOI: 10.3390/rs12244033.

2018

- Michelle Picoli, Gilberto Camara, et al., "Big Earth Observation Time Series Analysis for Monitoring Brazilian Agriculture". ISPRS Journal of Photogrammetry and Remote Sensing, 2018.

Introduction to SITS

How `sits` works

The `sits` package is based on the premise of using all of the data available in an Earth observation data cube, adopting a *time-first, space-later* approach. Each spatial location is associated to a time series. Locations with known labels train a machine learning classifier, which classifies all time series of a data cube, as shown in Figure 1.

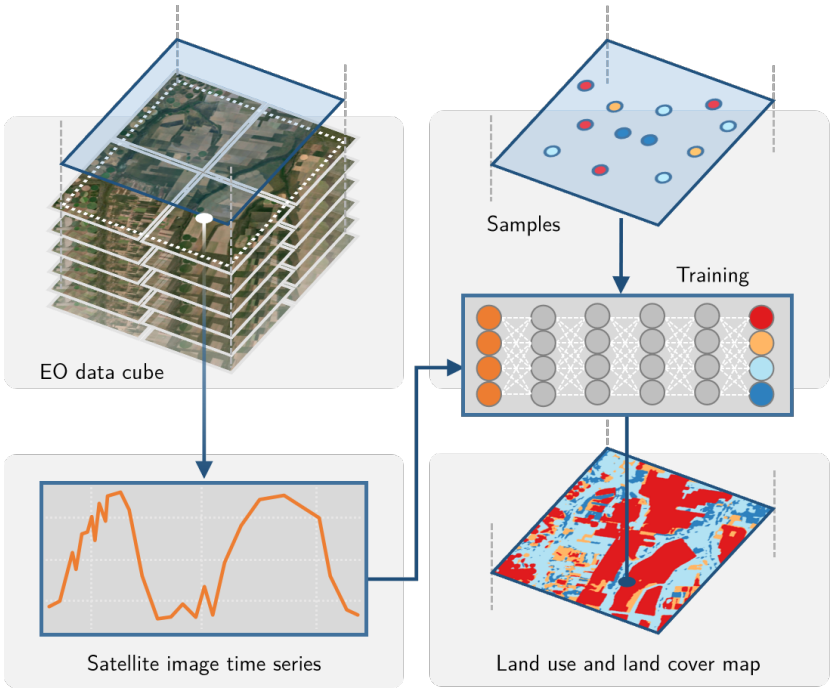


Figure 1: Using time series for land classification (source: authors)

The package provides a set of tools for analysis, visualization and classification of satellite image time series. Users follow a typical workflow:

1. Select an analysis-ready data image collection on cloud providers such as AWS, Microsoft Planetary Computer, Digital Earth Africa and Brazil Data Cube.
2. Build a regular data cube using the chosen image collection.
3. Obtain new bands and indices with operations on data cubes.
4. Extract time series samples from the data cube to be used as training data.
5. Perform quality control and filtering on the time series samples.
6. Train a machine learning model using the extracted samples.
7. Use the model to classify the data cube and get class probabilities for each pixel.
8. Post-process the probability cube to remove outliers.
9. Produce a labeled map from the post-processed probability cube.
10. Evaluate the accuracy of the classification using best practices.

Each step of workflow corresponds to a function of the `sits` API, as shown in the table and figure below. These functions have convenient default parameters and behavior. A single function builds machine learning models ranging from random forests to neural networks. Functions that process big data cubes implement efficient processing internally. Thus, users can achieve good results without in-depth knowledge about machine learning and parallel processing.

Table 2: The sits API workflow for land classification

API_function	Inputs	Output
`sits_cube()`	ARD image collection	Irregular data cube
`sits_regularize()`	Irregular data cube	Regular data cube
`sits_apply()`	Regular data cube	Regular data cube with new bands and indices
`sits_get_data()`	Data cube and sample locations	Time series
`sits_som()`	Time series	Improved time series samples
`sits_train()`	Time series and ML algorithm	ML classification model
`sits_classify()`	ML classification model and regular data cube	Probability cube
`sits_smooth()`	Probability cube	Post-processed probs cube
`sits_label_classification()`	Post-processed probs cube	Classified map
`sits_accuracy()`	Classified map and validation samples	Accuracy assessment

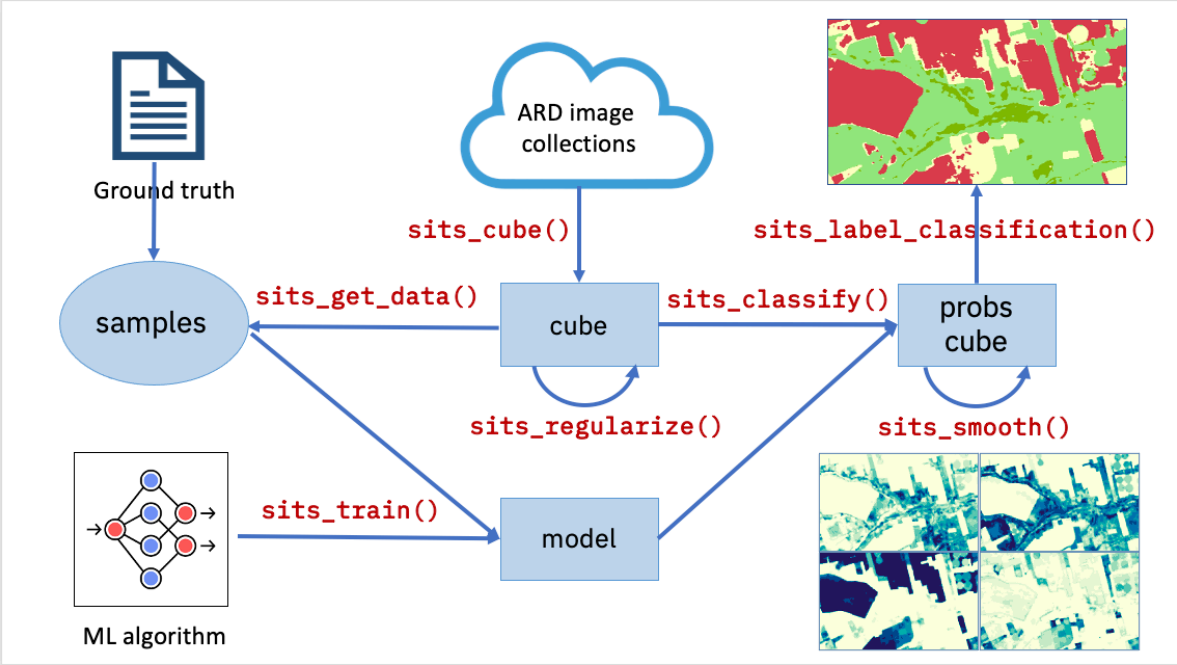


Figure 2: Main functions of the SITS API (source: authors).

Creating a Data Cube

There are two kinds of data cubes in `sits`: (a) irregular data cubes generated by selecting image collections image collections on cloud providers such as AWS and Planetary Computer; (b) regular data cubes with images fully covering a chosen area, where each image has the same spectral bands and spatial resolution, and images follow a set of adjacent and regular time intervals. Machine learning applications need regular data cubes. For further details, please refer to Chapter "Earth Observation Data Cubes".

The first steps in using `sits` are: (a) select an analysis-ready data image collection available in a cloud provider or stored locally using `sits_cube()`; (b)if the collection is not regular, use `sits_regularize()` to build a regular data cube.

This section shows how to build a data cube from images organized as a regular data cube and available as local files. The data cube is composed of a set of MODIS MOD13Q1 images for the Sinop region in Mato Grosso, Brazil. All images have bands "NDVI" and "EVI" covering a one-year period from 2013-09-14 to 2014-08-29 (we use "year-month-day" for dates). There are 23 time instances, each covering a 16-day period. The data is available in the R package `sitsdata`.

To build a data cube from local files, users need to provide information about the original source from with the data was obtained. In this case, `sits_cube()` needs the parameters:

(a) `source`, the cloud provider from where the data has been obtained (in this case the Brazil Data Cube "BDC");
(b) `collection`, the collection from where the images have been extracted. In this case, data comes from the MOD13Q1 collection 6;
(c) `data_dir`, the local directory where the image files are stored;
(d) `parse_info`, stating how file names store information on tile, band and date. In this case, local images stored as `TERRA_MODIS_012010_EVI_2014-07-28.tif`.

```
data_dir <- system.file("extdata/sinop", package = "sitsdata")
# create a data cube object based on the information about the files
sinop <- sits_cube(
  source = "BDC",
  collection = "MOD13Q1-6",
  data_dir = data_dir,
  parse_info = c("X1", "X2", "tile", "band", "date")
)
# plot a color composite for the first date (2013-09-14)
plot(sinop,
    red = "EVI",
    green = "NDVI",
    blue = "EVI",
    date = "2013-09-14"
)
```

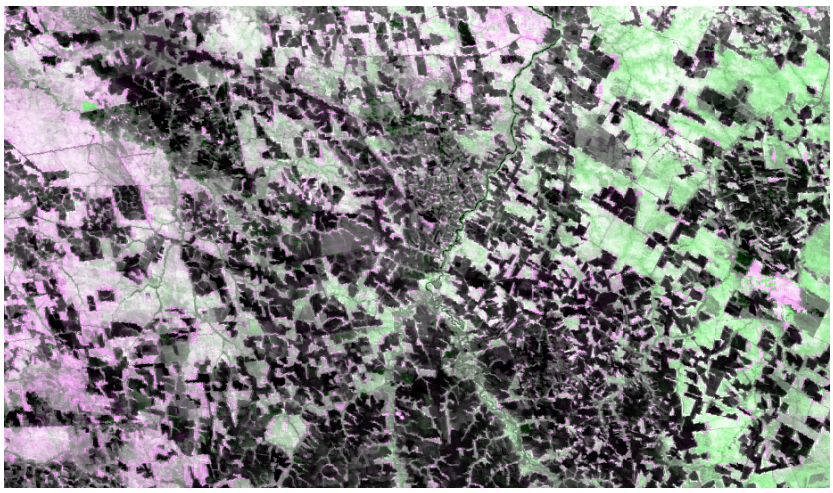


Figure 3: Color composite image MOIDS cube for 2013-09-14 (red = EVI, green = NDVI, blue = EVI).

The R object returned by `sits_cube()` contains the metadata that describes the the contents of the data cube. The metadata includes data source and collection, satellite, sensor, tile in the collection, bounding box, projection, and list of files. Each file refers to one band of an image at one of the temporal instances of the cube.

```
# show the R object that describes the data cube
sinop
```

```
#> # A tibble: 1 x 11
#>  source collection satellite sensor tile xmin
#>  <chr> <chr>   <chr>   <chr> <chr>   <dbl>
#> 1 BDC MOD13Q1-6 TERRA MODIS 012010 -6181982.
#> # ... with 5 more variables: xmax <dbl>, ymin <dbl>,
#> # ymax <dbl>, crs <chr>, file_info <list>
```

The time series table

To handle time series information, `sits` uses a tabular data structure. The example below shows a table with 1,218 time series obtained from MODIS MOD13Q1 images. Each series has four attributes: two bands ("NIR" and "MIR") and two indexes ("NDVI" and "EVI"). This data set is available in package `sitsdata`.

```
# load the MODIS samples for Mato Grosso from the
# 'sitsdata' package
library(sitsdata)
data("samples_matogrosso_mod13q1", package = "sitsdata")
samples_matogrosso_mod13q1[1:2, ]
```

```
#> # A tibble: 2 x 7
#>  longitude latitude start_date end_date label cube
#>     <dbl>  <dbl> <date> <date>  <chr> <chr>
#> 1   -57.8  -9.76 2006-09-14 2007-08-29 Pasture bdc_~
#> 2   -59.4  -9.31 2014-09-14 2015-08-29 Pasture bdc_~
#> # ... with 1 more variable: time_series <list>
```

The data structure associated to the time series is a table that contains data and metadata. The first six columns contain the metadata: spatial and temporal information, the label assigned to the sample, and the data cube from where the data has been extracted. The `time_series` column contains the time series data for each spatiotemporal location. This data is also organized as a table, with a column with the dates and the other columns with the values for each spectral band. For more details on how to handle time series data, please see the "Working with Time Series" chapter.

It is useful to plot the dispersion of the time series. In what follows, for brevity we will select only one label ("Forest") and one index ("NDVI"). The resulting plot shows all of the time series associated to the label and attribute, highlighting the median and the first and third quartiles.

```
samples_forest <- dplyr::filter(
   samples_matogrosso_mod13q1,
   label == "Forest"
)
samples_forest_ndvi <- sits_select(
   samples_forest,
   band = "NDVI"
)
plot(samples_forest_ndvi)
```

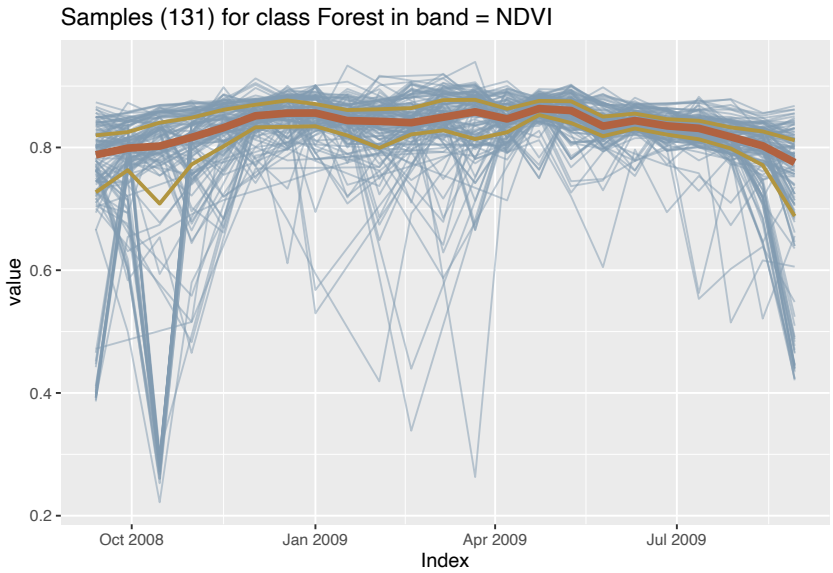


Figure 4: Joint plot of all samples in band NDVI for class Forest.

Training a machine learning model

The next step is to train a machine learning (ML) model using `sits_train()`. It takes two inputs, `samples` (a time series table) and `ml_method` (a function that implements a ML algorithm). The result is a model that is used classification. Each algorithm requires specific parameters that are user-controllable. For novice users, `sits` provides default parameters which produces a good result. For more details, please see the "Machine Learning for Data Cubes" chapter.

Since the time series data has four attributes ("EVI", "NDVI", "NIR", "MIR") and the data cube images only two, we select the "NDVI" and "EVI" values and use the resulting data for training. To build the classification model, we use a random forest model called by the `sits_rfor()` function.

```
# select the bands "ndvi", "evi"
samples_2bands <- sits_select(
   data = samples_matogrosso_mod13q1,
   bands = c("NDVI", "EVI"))

# train a random forest model
rf_model <- sits_train(
   samples = samples_2bands,
   ml_method = sits_rfor()
)
# plot the most important variables of the model
plot(rf_model)
```

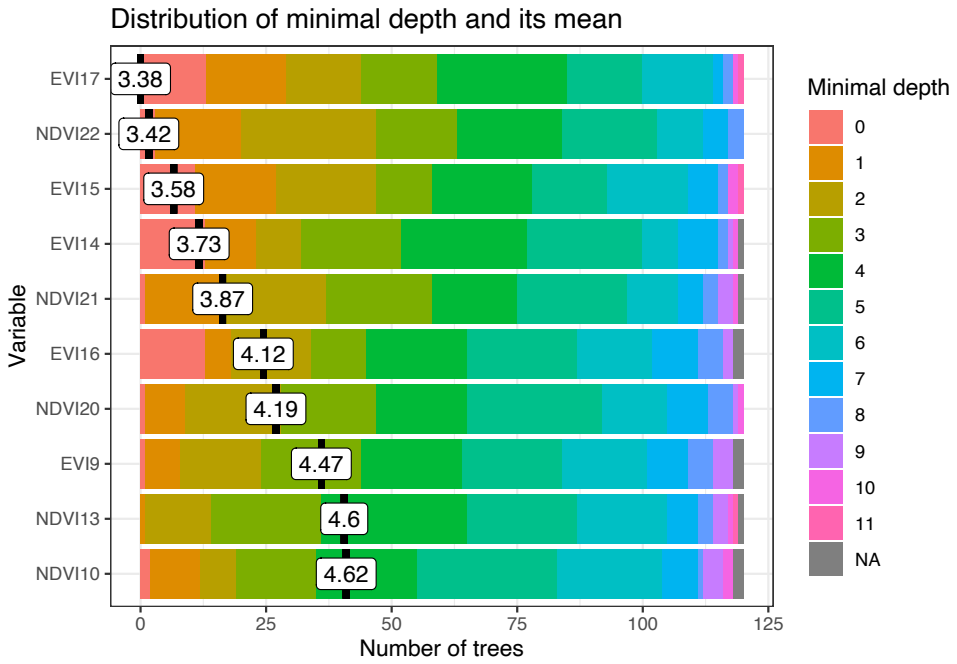


Figure 5: Most relevant variables of trained random forests model.

Data cube classification

After training the machine learning model, the next step is to classify the data cube using `sits_classify()`. This function produces a set of raster probability maps, one for

each class. For each of these maps, the value of a pixel is proportional to the the probability that it belongs to the class. This function has two mandatoty parameters: `data`, the data cube or time series tibble to be classified; and `ml_model`, the trained ML model. Optional parameters include: (a) `multicores`, number of cores to be used; (b) `memsize`, RAM used in the classification; (c) `output_dir`, the directory where the classified files will be written. Details of the classification process are available in "Classification of Images in Data Cubes".

```
# classify the raster image
sinop_probs <- sits_classify(
   data = sinop,
   ml_model = rf_model,
   multicores = 2,
   memsize = 8,
   output_dir = "./tempdir/chp3"
)
# plot the probability cube
plot(sinop_probs, labels = c("Forest"))
```

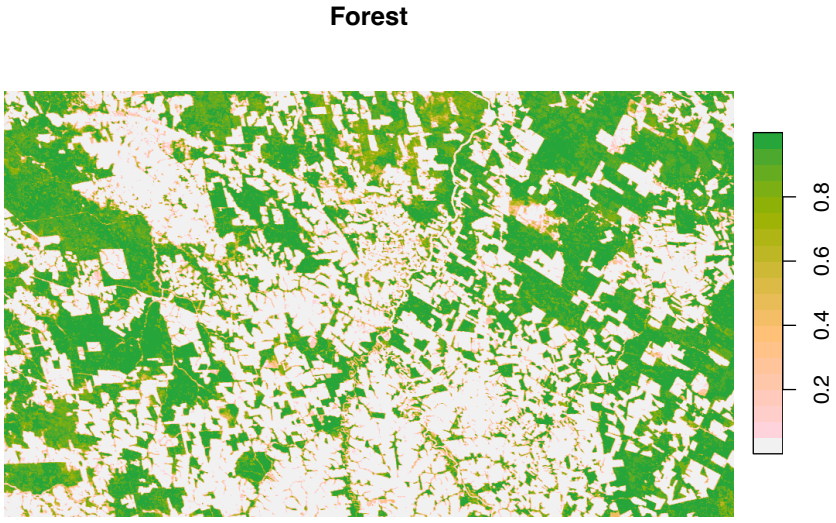


Figure 6: Probability map for class Forest.

After classification has been completed, we plot the probability maps for class "Forest". Probability maps are useful to visualize the degree of confidence that the classifier assigns to the labels for each pixel and can be used to produce uncertainty information and support active learning, as described in Chapter "Data Cube Classification".

Spatial smoothing

When working with big EO data, there is much variability in each class. As a result, some pixels will be misclassified. These errors are more likely to occur in transition areas between classes. To offset these problems, the `sits_smooth()` function takes a probability cube as input and uses the class probabilities of each pixel's neighborhood to reduce labeling uncertainty. We then plot the smoothed probability map for class "Forest" to compare with the previous plot.

```
# perform spatial smoothing
sinop_bayes <- sits_smooth(
   cube = sinop_probs,
   multicores = 2,
   memsize = 8,
   output_dir = "./tempdir/chp3"
)
plot(sinop_bayes, labels = c("Forest"))
```

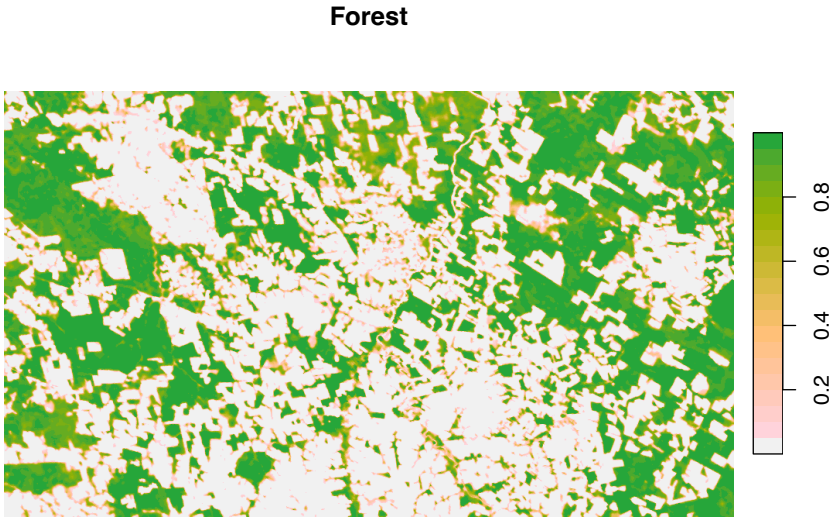


Figure 7: Smoothed probability map for class Forest.

Labelling a probability data cube

After removing outliers using local smoothing, one can obtain the labeled classification map using the function `sits_label_classification()`. This function assigns each pixel to the class with highest probability.

```
# label the probability file
sinop_map <- sits_label_classification(
   cube = sinop_bayes,
   output_dir = "./tempdir/chp3"
)
plot(sinop_map, title = "Sinop Classification Map")
```

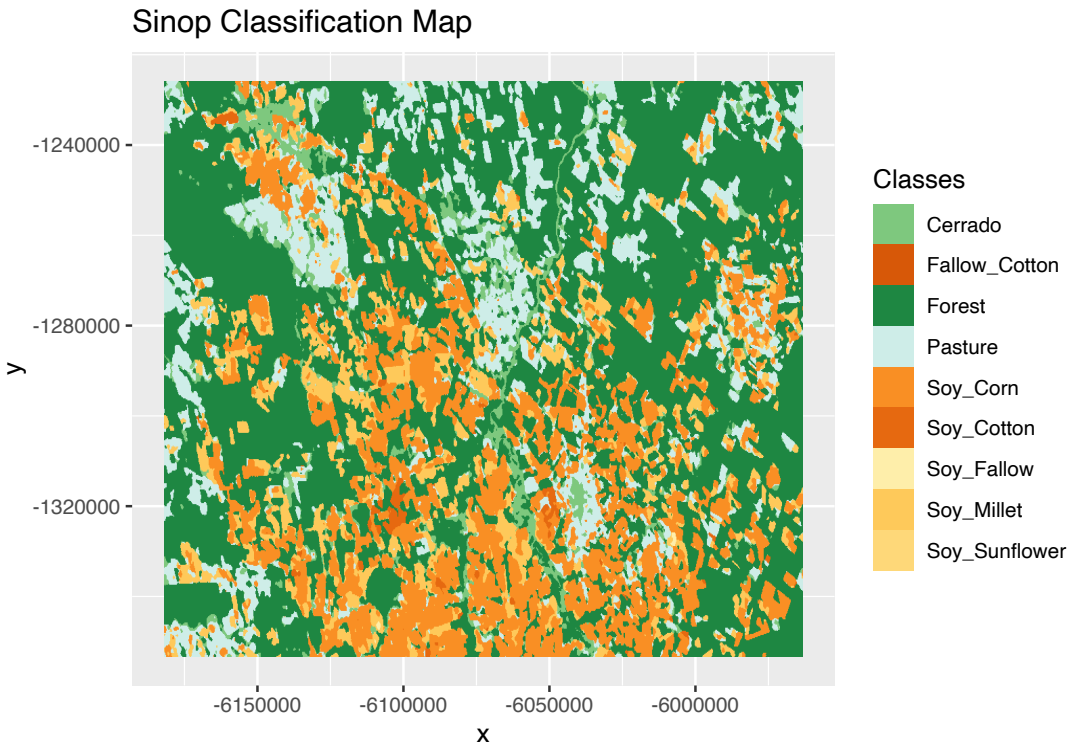


Figure 8: Classification map for Sinop

In the above example, the color legend is taken for the predefined color table provided by `sits`. The options for defining labels include:

1. Predefined color table: `sits` has a default color table, that can be shown using the command `sits_config_show(colors = TRUE)`. This color definition file assigns colors to about 250 class names, including the IPCC and IGBP land use classes, the UMD GLAD classification scheme, the FAO LCC1 and LCCS2 land use layers, and the LCCS3 surface hydrology layer.
2. User-defined defined color table: users can set the legend they prefer with a YAML user-defined configuration file. This file should defined by the environmental variable `SITS_CONFIG_USER_FILE`. Create an YAML file, and then set the path to it with `Sys.setenv(SITS_CONFIG_USER_FILE="path_to_myfile")`. An example of an YAML file with user-defined legend is shown below.

3. User-defined legend: users may define their own legends and pass them as parameters to to the `plot` function.

```
colors:
   Cropland:      "khaki"
   Deforestation: "sienna"
   Forest :       "darkgreen"
   Grassland :    "lightgreen"
   NonForest:     "lightsteelblue1"
```

The resulting classification files can be read by QGIS. Links to the associated files are available in the `sinop_map` object in the nested table `file_info`.

```
# show the location of the classification file
sinop_map$file_info[[1]]
```

```
#> # A tibble: 1 x 12
#>  band start_date end_date xmin   ymin  xmax
#>  <chr> <date> <date>     <dbl>  <dbl> <dbl>
#> 1 class 2013-09-14 2014-08-29 -6181982. -1.35e6 -5.96e6
#> # ... with 6 more variables: ymax <dbl>, xres <dbl>,
#> # yres <dbl>, nrows <dbl>, ncols <dbl>, path <chr>
```

Final remarks

The `sits` package provides an API to build EO data cubes from image collections available in cloud services, and to perform land classification of data cubes using machine learning. The classification models are built based on satellite image time series extracted from the cubes. The package provides additional function for sample quality control, post-processing and validation. The design of the API tries to reduce complexity for users and hide details such as how to do parallel processing, and to handle data cubes composed by tiles of different timelines.

Earth observation data cubes

Analysis-ready data image collections

Collections of Earth observation analysis-ready data (ARD) are available in cloud services such as Amazon Web Service, Brazil Data Cube, Digital Earth Africa, Swiss Data Cube, and Microsoft's Planetary Computer. These collections which have been processed to improve multidate comparability. Radiance measures at the top of the atmosphere have been converted to ground reflectance measures. In general, timelines of the images of an ARD image collection are different. Images still contain cloudy or missing pixels; bands for the images in the collection may have different resolutions. Figure 9 shows an example of the Landsat ARD image collection.

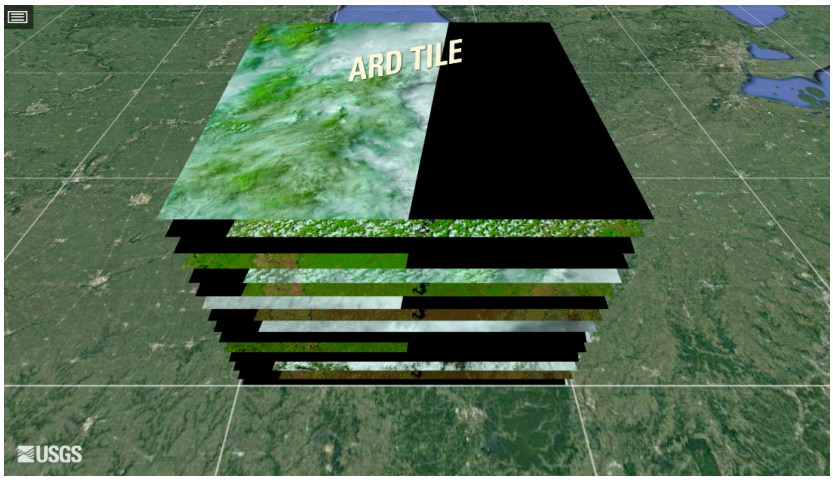


Figure 9: ARD image collection (source: USGS). Reproduction based on fair use doctrine.

ARD image collections are organized in spatial partitions. Sentinel-2/2A images follow the MGRS tiling system, which divides the world in 60 UTM zones of 8 degrees of longitude each. Each zone has blocks of 6 degrees of latitude. Blocks are split into tiles of 110 x 110 km2 with a 10 km overlap. Figure 10 shows the MGRS tiling system for a part of the Northeastern coast of Brazil, contained in UTM zone 24, block M.

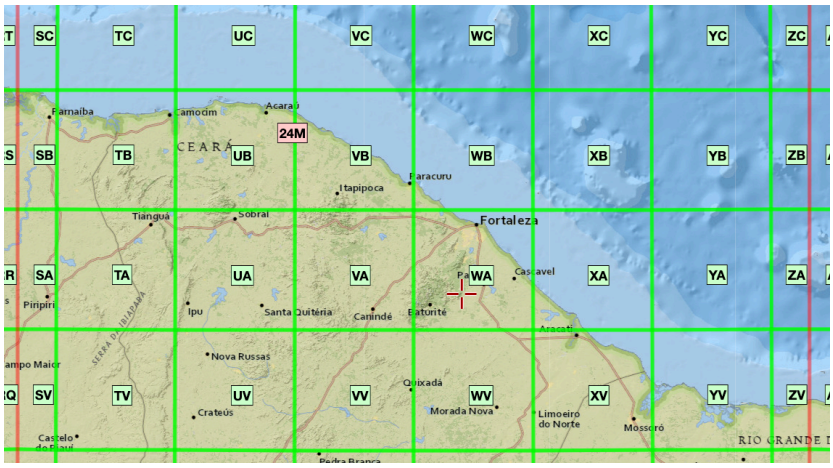


Figure 10: MGRS tiling system used by Sentinel-2 images (source: GISSurfer 2.0). Reproduction based on fair use doctrine.

The Landsat 4/5/7/8/9 satellites use the Worldwide Reference System (WRS-2), which breaks the coverage of Landsat satellites into images identified by path and row. The path is the descending orbit of the satellite; the WRS-2 system has 233 paths per orbit and each path is divided in 119 rows,where each row refers to ta latitudinal center line of a frame of imagery. Images in WRS-2 are geometrically corrected to the UTM projection.

Figure 11: WRS-2 tiling system used by Landsat-5/7/8/9 images (source: INPE and ESRI). Reproduction based on fair use doctrine.

ARD image collections handled by sits

As of version 1.1.0, `sits` supports access to the following ARD image collections:

1. Amazon Web Services (AWS): Open data Sentinel-2/2A level 2A collections for the Earth's land surface.
2. Brazil Data Cube (BDC): Open data collections of Sentinel-2/2A, Landsat-8, CBERS-4/4A, and MODIS images for Brazil. These collections organized as regular data cubes.
3. Digital Earth Africa (DEA): Open data collections of Sentinel-2/2A and Landsat-8 for Africa.
4. Microsoft Planetary Computer (MPC): Open data collections of Sentinel-2/2A and Landsat-4/5/7/8/9 for the Earth's land areas.
5. USGS: Landsat-4/5/7/8/9 collections available in AWS, which require payment to access.
6. Swiss Data Cube (SDC): Open data collection of Sentinel-2/2A and Landsat-8 images for Switzerland.

Regular image data cubes

Machine learning and deep learning (ML/DL) classification algorithms require the input data to be consistent. The dimensionality of the data used for training the model has to be the same as that of the data to be classified. There should be no gaps in the input data and no missing values are allowed. Thus, to use of ML/DL algorithms for remote sensing data, ARD image collections should be converted to regular data cubes. Following [2], a *regular data cube* meets the following definition:

1. A regular data cube is a four-dimensional structure with dimensions x (longitude or easting), y (latitude or northing), time, and bands.
2. Its spatial dimensions refer to a single spatial reference system (SRS). Cells of a data cube have a constant spatial size with respect to the cube's SRS.
3. The temporal dimension is composed of a set of continuous and equally-spaced intervals.
4. For every combination of dimensions, a cell has a single value.

All cells of a data cube have the same spatiotemporal extent. The spatial resolution of each cell is the same in X and Y dimensions. All temporal intervals are the same. Each cell contains a valid set of measures. For each position in space, the data cube should provide a valid time series. For each time interval, the regular data cube should provide a valid 2D image (see Figure 11).

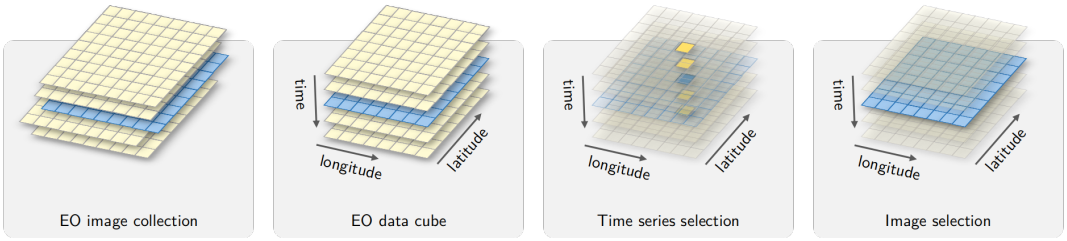


Figure 12: Conceptual view of data cubes (source: authors)

Currently, the only cloud service that provides regular data cubes by default is the Brazil Data Cube (BDC). Analysis-ready data (ARD) collections available in AWS, MSPC, USGS and DE Africa are not regular in space and time. Bands may have different resolutions, images may not cover the entire time, and time intervals are not regular. For this reason, subsets of these collections need to be converted to regular data cubes before further processing. To produce data cubes for machine-learning data analysis, users should first create an irregular data cube from an ARD collection and then use `sits_regularize()`, as described below.

Creating data cubes

To obtain information on ARD image collection from cloud providers, `sits` uses the STAC (SpatioTemporal Asset Catalogue) protocol, which is a specification of geospatial information which has been adopted by many large image collection providers. A 'spatiotemporal asset' is any file that represents information about the Earth captured in a certain space and time. To access STAC endpoints, `sits` uses the rstac[12] R package.

All access to image collections and data cubes is done using `sits_cube()`, which requires the following common parameters:

1. `source`: name of the provider from the list of providers supported by `sits`.
2. `collection`: a collection available in the provider and supported by `sits`. To find out which collections are supported by `sits`, see `sits_list_collections()`.
3. `platform`: optional parameter specifying the platform in case of collections that include more than one satellite.
4. `tiles`: Set of tiles of image collection reference system. Either `tiles` or `roi` should be specified.
5. `roi`: a region of interest. Either: (a) a named vector (`lon_min`, `lon_max`, `lat_min`, `lat_max`) in WGS 84 coordinates; or (b) an `sf` object. All images that intersect the convex hull of the `roi` are selected.

[12] http://github.com/brazil-data-cube/rstac

6. `bands`: (optional) bands to be used. If missing, all bands from the collection are used.
7. `start_date`: the initial date for the temporal interval containing the time series of images.
8. `end_date`: the final date for the temporal interval containing the time series of images.

The `sits_cube()` function produces a tibble with metadata about the desired images. It has the information required for further processing, but does not contain the actual data. The attributes of individual image files can be assessed by listing the `file_info` column of the tibble.

Assessing Amazon Web Services

AWS holds are two kinds of collections: *open-data* and *requester-pays*. Open data collections can be accessed without cost. Requester-pays collections require payment to an AWS account. Currently, `sits` supports collections `SENTINEL-S2-L2A` (requester-pays) and `SENTINEL-S2-L2A-COGS` (open-data). Both collections include all Sentinel-2/2A bands. The bands in 10m resolution are `B02`, `B03`, `B04`, and `B08`. The 20m bands are `B05`, `B06`, `B07`, `B8A`, `B11`, and `B12`. Bands `B01` and `B09` are available at 60m resolution. A `CLOUD` band is also available. The example below shows how to access one tile of the open data `SENTINEL-S2-L2A-COGS` collection. The `tiles` parameter allows selection of desired area according to the MGRS reference system.

```
# create a data cube covering an area in the Brazilian Amazon
# Sentinel-2 images over the Rondonia region
s2_20LKP_cube <- sits_cube(
   source = "AWS",
   collection = "SENTINEL-S2-L2A-COGS",
   start_date = "2018-07-12",
   end_date = "2019-07-28",
   tiles = "20LKP",
   bands = c("B04", "B08", "B11", "CLOUD")
)
```

Assessing Microsoft's Planetary Computer

Microsoft's Planetary Computer (MPC) hosts two open data collections: `SENTINEL-2-L2A` and `LANDSAT-C2-L2`. The first collection contains SENTINEL-2/2A ARD images, with the same bands and resolutions as those available in AWS (see above). The example below shows how to access the `SENTINEL-2-L2A` collection.

```r
# create a data cube covering an area in the Brazilian Amazon
s2_20LKP_cube_MPC <- sits_cube(
     source = "MPC",
     collection = "SENTINEL-2-L2A",
     tiles = "20LKP",
     start_date = "2019-07-01",
     end_date = "2019-07-28",
     bands = c("B05", "B8A", "B11", "CLOUD")
)
# plot a color composite of one date of the cube
plot(s2_20LKP_cube_MPC, red = "B11", blue = "B05", green = "B8A",
    date = "2019-07-18")
```

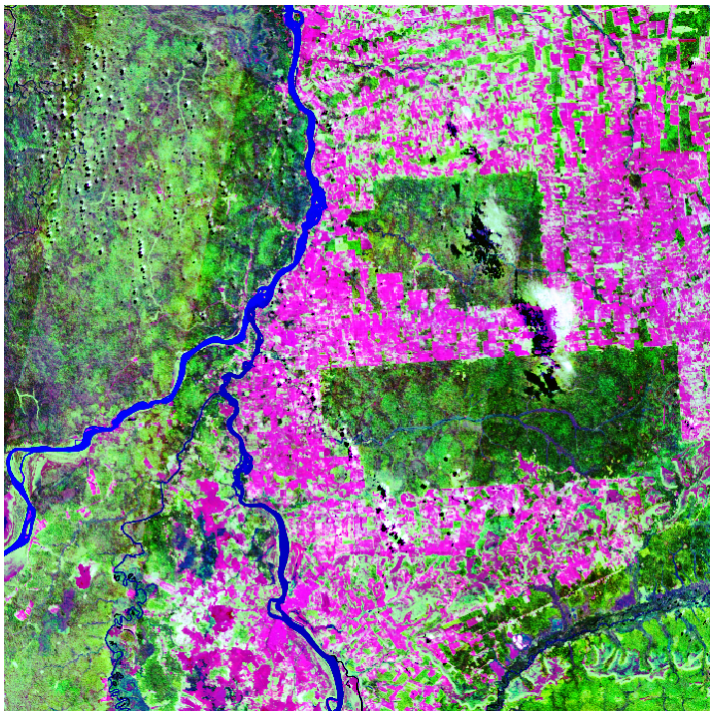


Figure 13: Sentinel-2 image in an area of the state of Rondonia, Brazil

The `LANDSAT-C2-L2` collection provides access to data from Landsat-4, 5, 7, 8, and 9 satellites. Images from these satellites have been intercalibrated to ensure data consistency. For compatibility between the different Landsat sensors, the band names are `BLUE`, `GREEN`, `RED`, `NIR08`, `SWIR16`, and `SWIR22`. All images have 30m resolution. For this collection, tile search is not supported; the `roi` parameter should be used. The example shows how to retrieve data from a region of interest covering the city of Brasilia in Brazil.

```r
# Read a shapefile thar covers the city of Brasilia
shp_file <- system.file("extdata/shapefiles/df_bsb/df_bsb.shp",
                  package = "sits")
sf_bsb <- sf::read_sf(shp_file)
# select the cube
s2_L8_cube_MPC <- sits_cube(
      source = "MPC",
      collection = "LANDSAT-C2-L2",
      roi = sf_bsb,
      start_date = "2019-06-01",
      end_date = "2019-10-01",
      bands = c("BLUE", "NIR08", "SWIR16", "CLOUD")
)
# Plot the second tile that covers Brasilia
plot(s2_L8_cube_MPC[2,], red = "SWIR16", green = "NIR08", blue = "BLUE",
    date = "2019-07-30")
```

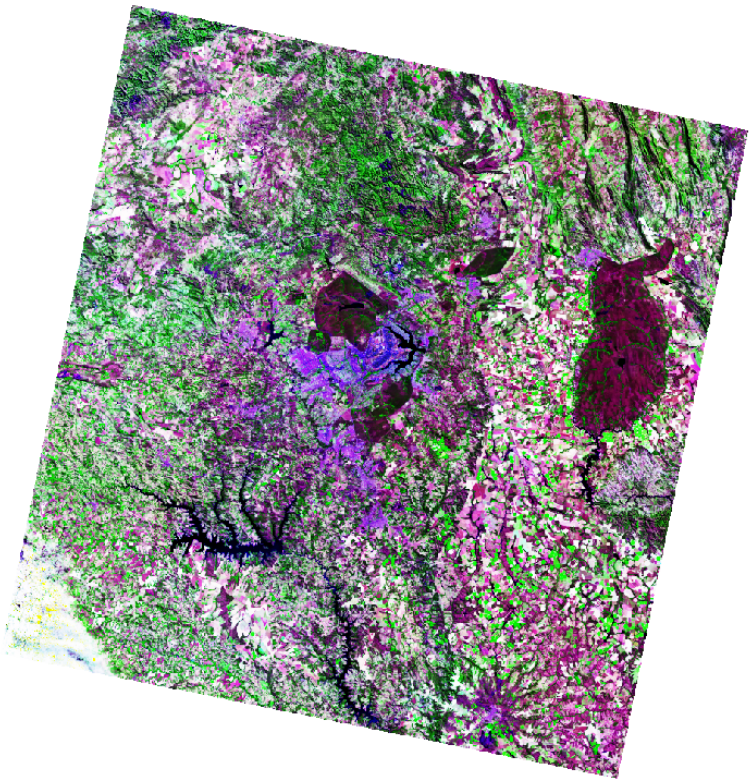


Figure 14: Landsat-8 image in an area of the city of Brasilia, Brazil

Assessing Digital Earth Africa

Digital Earth Africa (DEAFRICA) is a cloud service that provides open access Earth Observation data for the African continent. The ARD image collections available in `sits` are `S2_L2A` (Sentinel-2 level 2A) and `LS8_SR` (Landsat-8). Since the STAC interface for DEAFRICA does not implement the concept of tiles, users users need to specify their area of interest using the `roi` parameter. The requested `roi` produces a cube that contains three MGRS tiles ("35HLD", "35HKD", and "35HLC") covering part of South Africa.

```
dea_cube <- sits_cube(
   source = "DEAFRICA",
   collection = "S2_L2A",
   bands = c("B05", "B8A", "B11"),
   roi = c(lon_min = 24.97, lat_min = -34.30,
         lon_max = 25.87, lat_max = -32.63),
   start_date = "2019-09-01",
   end_date = "2019-10-01"
)
# plot tile 35HLC
dea_cube %>%
   dplyr::filter(tile == "35HLC") %>%
   plot(red = "B11", blue = "B05", green = "B8A", date = "2019-09-07")
```

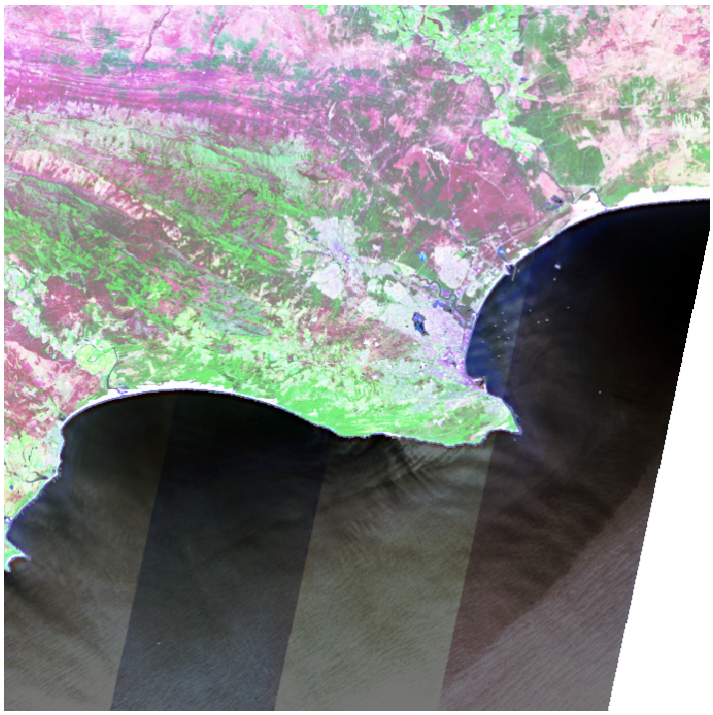


Figure 15: Sentinel-2 image in an area over South Africa

Assessing the Brazil Data Cube

The Brazil Data Cube[13] (BDC) is built by Brazil's National Institute for Space Research (INPE). The BDC uses three hierarchical grids based on the Albers Equal Area projection and SIRGAS 2000 datum. The three grids are generated taking -54° longitude as the central reference and defining tiles of 6×4, 3×2 and 1.5×1 degrees. The large grid is composed by tiles of 672×440 km^2 and is used for CBERS-4 AWFI collections at 64 meter resolution; each CBERS-4 AWFI tile contains images of $10,504 \times 6,865$ pixels. The medium grid is used for Landsat-8 OLI collections at 30 meter resolution; tiles have an extension of 336×220 km^2 and each image has $11,204 \times 7,324$ pixels. The small grid covers 168×110 km^2 and is used for Sentinel-2 MSI collections at 10m resolutions; each image has $16,806 \times 10,986$ pixels. The data cubes in the BDC are regularly spaced in time and cloud-corrected [3].

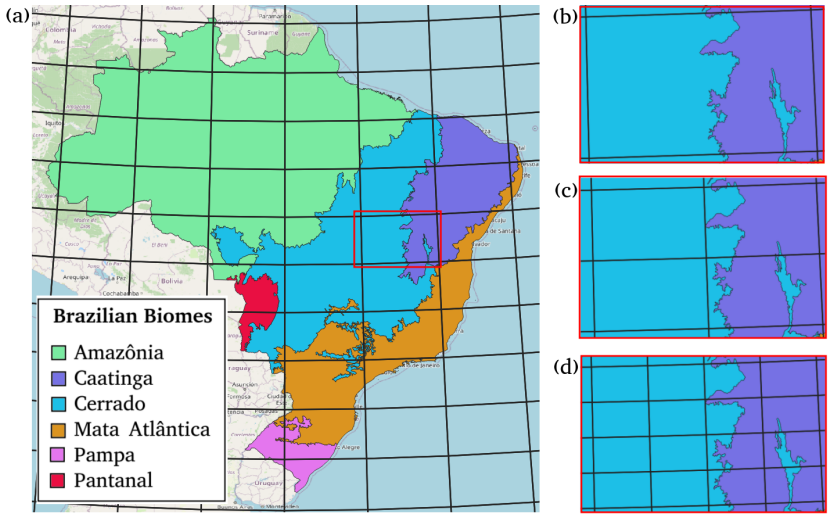


Figure 16: Hierarchical BDC tiling system showing overlayed on Brazilian Biomes (a), illustrating that one large tile (b) contains four medium tiles (c) and that medium tile contains four small tiles. Source: Ferreira et al.(2020). Reproduction under fair use doctrine.

The collections available in the BDC are: `LC8_30_16D_STK-1` (Landsat-8 OLI, 30m resolution, 16-day intervals), `S2-SEN2COR_10_16D_STK-1` (Sentinel-2 MSI images at 10 meter resolution, 16-day intervals), `CB4_64_16D_STK-1` (CBERS 4/4A AWFI, 64m resolution, 16 days intervals), `CB4_20_1M_STK-1` (CBERS 4/4A MUX, 20m resolution, one month intervals) and `MOD13Q1-6` (MODIS MOD13SQ1 product, collection 6, 250m resolution, 16-day intervals). For more details, use `sits_list_collections(source = "BDC")`.

[13] http://brazildatacube.org/

To access the Brazil Data Cube, users need to provide their credentials using an environment variables, as shown below. Obtaining a BDC access key is free. Users need to register at the BDC site[14] to obtain the key.

```
Sys.setenv(
   "BDC_ACCESS_KEY" = <your_bdc_access_key>
)
```

In the example below, the data cube is defined as one tile ("022024") of `CB4_64_16D_STK-1` collection which holds CBERS AWFI images at 16 days resolution.

```
# define a tile from the CBERS-4/4A AWFI collection
cbers_tile <- sits_cube(
   source = "BDC",
   collection = "CB4_64_16D_STK-1",
   bands = c("B13", "B14", "B15", "B16", "CLOUD"),
   tiles = "022024",
   start_date = "2018-09-01",
   end_date = "2019-08-28"
)
# plot one time instance
plot(cbers_tile, red = "B15", green = "B16", blue = "B13", date = "
    2018-09-30")
```

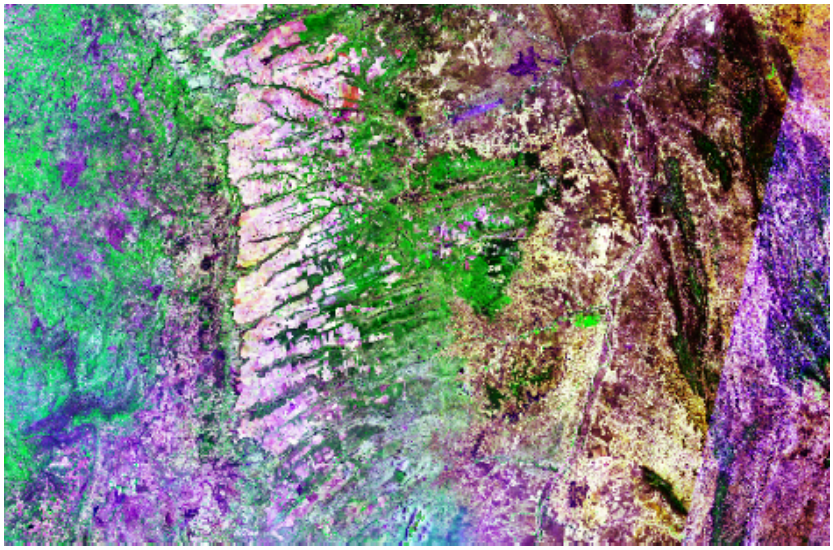


Figure 17: Plot of CBERS-4 image obtained from the BDC with a single tile covering an area in the Brazilian Cerrado.

[14] https://brazildatacube.dpi.inpe.br/portal/explore

Defining a data cube using ARD local files

ARD images downloaded from cloud collections to a local computer are not associated to a STAC endpoint that describes them. They need to be be organized and named to allow `sits` to create a data cube from them. All local files should be in the same directory and have the same spatial resolution and projection. Each file should contain a single image band for a single date. Each file name needs to include tile, date and band information. Users need to provide information about the original data source to allow `sits` to retrieve information about image attributes such as band names, missing values, etc. When working with local cubes, `sits_cube()` needs the following parameters:

1. `source`: name of the original data provider; either `BDC`, `AWS`, `USGS`, `MSPC` or `DEAFRICA`.
2. `collection`: collection from where the data was extracted.
3. `data_dir`: local directory for images.
4. `bands`: optional parameter to describe the bands to be retrieved.
5. `parse_info`: information to parse the file names. File names need to contain information on tile, date and band, separated by a delimiter (usually "_").
6. `delim`: separator character between descriptors in the file name (default is "_").

The example shows how to define a data cube using files from the *sitsdata* package. Given the file name `CB4_64_16D_STK_022024_2018-08-29_2018-09-13_EVI.tif`, to retrieve information about the images, one needs to set the `parse_info` parameter to `c( "X1", "X2", "X3", "X4", "tile", "date", "X5", "band")`.

```
library(sits)
# Create a cube based on a stack of CBERS data
data_dir <- system.file("extdata/CBERS", package = "sitsdata")
# list the first file
list.files(data_dir)[1]
```

```
#> [1] "CB4_64_16D_STK_022024_2018-08-29_2018-09-13_EVI.tif"
```

```
# create a data cube from local files
cbers_cube <- sits_cube(
   source = "BDC",
   collection = "CB4_64_16D_STK-1",
   data_dir = data_dir,
   parse_info = c("X1", "X2", "X3", "X4", "tile", "date", "X5", "band")
)
# plot the band NDVI in the first time instance
plot(cbers_cube, band = "NDVI", date = "2018-08-29")
```

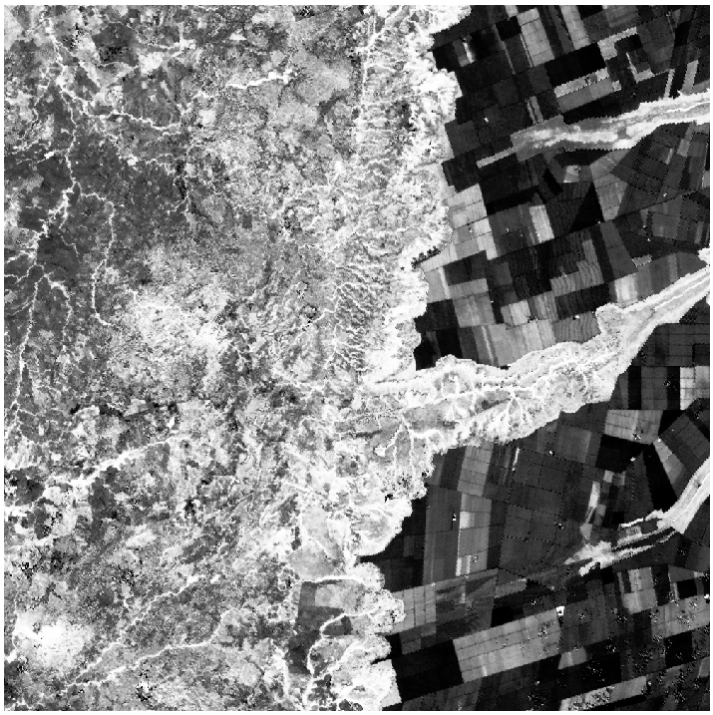


Figure 18: CBERS-4 NDVI in an area over Brazil

Defining a data cube using classified images

It is also possible to create local cubes based on results that have been produced by classification or post-classification algorithms. In this case, there are more parameters that are required and the parameter `parse_info` is specified differently, as follows:

1. `source`: name of the original data provider.
2. `collection`: name of the collection from where the data was extracted.
3. `data_dir`: local directory for the classified images.
4. `band`: Band name is associated to the type of result. Use `probs` for probability cubes produced by `sits_classify()`; `bayes`, `bilateral`, `gaussian` according to the function selected when using `sits_smooth()`; `entropy` when using `sits_uncertainty()`, `class` for labelled cubes.
5. `labels`: Labels associated to the classification results.
6. `version`: Version of the result (default = `v1`).
7. `parse_info`: File name parsing information to allow `sits` to deduce the values of `tile`, `start_date`, `end_date`, `band` and `version` from the file name. Unlike non-classified image files, cubes produced by classification and post-classification have both `start_date` and `end_date`.

The following code creates a results cube based on the classification of deforestation in Brazil. This classified cube was obtained by a large data cube of Sentinel-2 images, covering the state of Rondonia, Brazil comprising 40 tiles, 10 spectral bands, and covering the period from 2020-06-01 to 2021-09-11. Samples of four classes were trained by a random forest classifier.

```
# Create a cube based on a classified image
data_dir <- system.file("extdata/Rondonia", package = "sitsdata")
# file is named "SENTINEL-2_MSI_20LLP_2020-06-04_2021-08-26_class_v1.tif"
Rondonia_class_cube <- sits_cube(
   source = "AWS",
   collection = "SENTINEL-S2-L2A-COGS",
   data_dir = data_dir,
   bands = "class",
   labels = c("Burned_Area", "Cleared_Area",
            "Highly_Degraded", "Forest"),
   parse_info = c("X1", "X2", "tile", "start_date", "end_date",
               "band", "version")
)
# plot the classified cube
plot(Rondonia_class_cube)
```

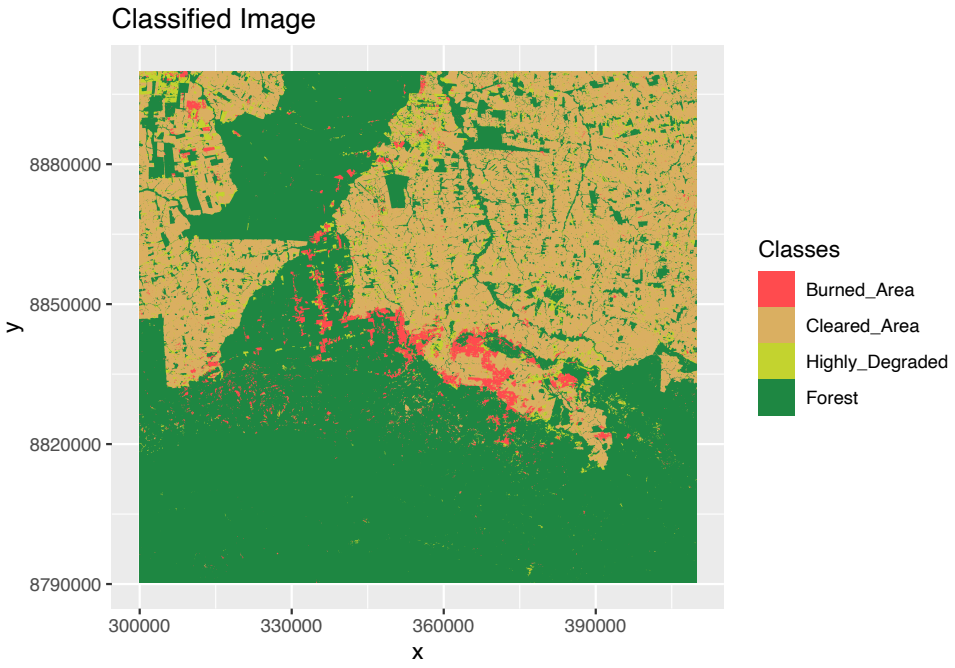


Figure 19: Classified data cube for year 2020/2021 in Rondonia, Brazil

Regularizing data cubes

Analysis-ready data (ARD) collections available in AWS, MSPC, USGS and DEAFRICA are not regular in space and time. Bands may have different resolutions, images may not cover the entire tile, and time intervals are not regular. For this reason, subsets of these collection need to be converted to regular data cubes before further processing and data analysis. This is done in *sits* by the function `sits_regularize()`, which uses the *gdalcubes* package [2]. In the following example, the user has created an irregular data cube from the Sentinel-2 collection available in Microsoft's Planetary Computer (MSPC) for tiles `20LKP` and `20LLP` in the state of Rondonia, Brazil. As described earlier in this chapter, because of the way ARD image collections are built, some of the images have invalid pixels, as shown below. We first build an irregular data cube using `sits_cube()`.

```
# creating an irregular data cube from MSPC
s2_cube <- sits_cube(
   source = "MPC",
   collection = "SENTINEL-2-L2A",
   tiles = c("20LKP", "20LLP"),
   start_date = as.Date("2018-07-01"),
   end_date = as.Date("2018-08-31"),
   bands = c("B05", "B8A", "B12", "CLOUD")
)
# plot the first image of the irregular cube
s2_cube %>%
   dplyr::filter(tile == "20LLP") %>%
   plot(red = "B12", green = "B8A", blue = "B05", date = "2018-07-03")
```

Because of different acquisition orbits of the Sentinel-2 and Sentinel-2A satellites, the two tiles also have different timelines. Tile `20LKP` has 12 instances and tile `20LLP` has 24 instances for the chosen period. The function `sits_regularize()` builds a data cube with a regular timeline and a best estimate of a valid pixel for each interval. The `period` parameter sets the time interval between two images. Values of `period` use the ISO8601 time period specification, which defines time intervals as `P[n]Y[n]M[n]D`, where Y stands for years, "M" for months and "D" for days. Thus, `P1M` stands for a one-month period, `P15D` for a fifteen-day period. When joining different images to get the best image for a period, `sits_regularize()` uses an aggregation method which organizes the images for the chosen interval in order of increasing cloud cover, and selects the first cloud-free pixel in the sequence.

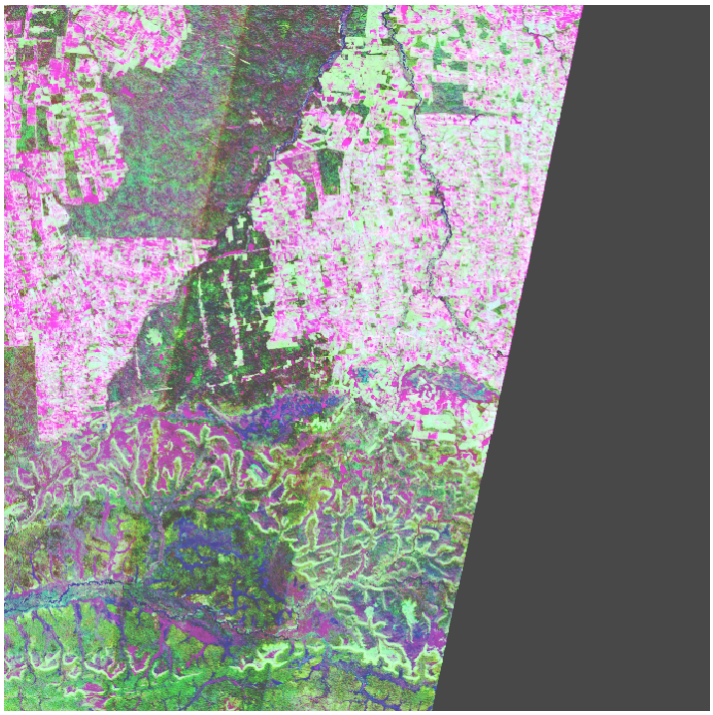


Figure 20: Sentinel-2 tile 20LLP for date 2018-07-03

```
# regularize the cube to 15 day intervals
reg_cube <- sits_regularize(
        cube    = s2_cube,
        output_dir = "./tempdir/chp4",
        res     = 120,
        period  = "P15D",
        multicores = 4,
        memsize = 16
)
# plot the first image of the tile 20LLP of the regularized cube
# The pixels of the regular data cube cover the full MGRS tile
reg_cube %>%
   dplyr::filter(tile == "20LLP") %>%
   plot(red = "B12", green = "B8A", blue = "B05")
```

After obtaining a regular data cube, users can perform data analysis and classification operations, as shown what follows and in the next chapter.

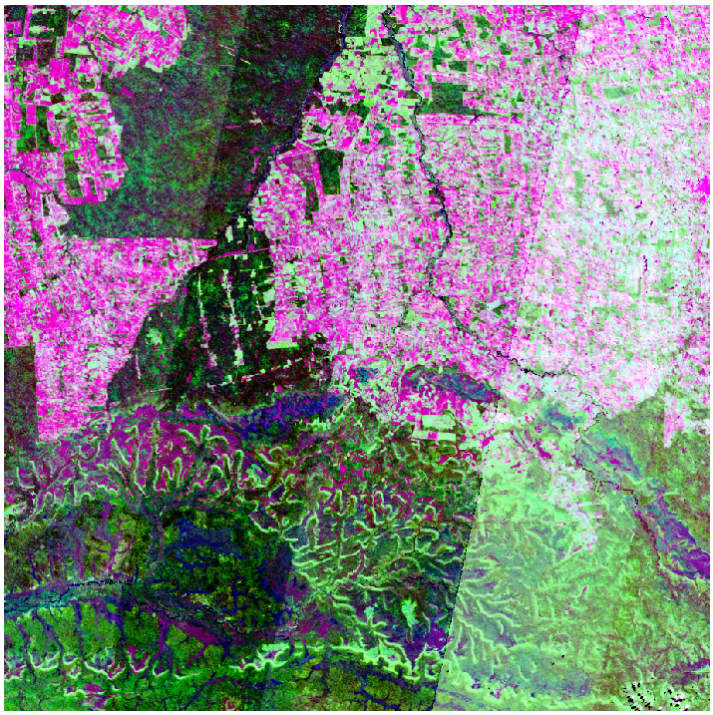


Figure 21: Regularized image for tile Sentinel-2 tile 20LLP

Mathematical operations on regular data cubes

Many analysis operations on remote sensing require operations on one or multiple bands. These operations include thresholding, decision trees, and band ratios. Given a regular data cube, these operations can be performed using `sits_apply()`. Users specify the mathematical operation as function of the bands available on the cube. The function then performs the operation for all tiles and all temporal intervals. The flexibility of `sits_apply()` allows users to perform different types of mathematical operations in data cubes in a simple and efficient fashion. The example below shows the computation of the normalized burn ratio (NBR) as the difference between the near infrared and the short wave infrared band, here calculated using the `B8A` and `B12` bands.

```
# calculate the normalized burn ratio
reg_cube_NBR <- sits_apply(reg_cube,
   NBR = (B8A - B12)/(B8A + B12),
   output_dir = "./tempdir/chp4",
   multicores = 4,
   memsize = 12
)
# plot the NBR for the first date
plot(reg_cube_NBR, band = "NBR")
```

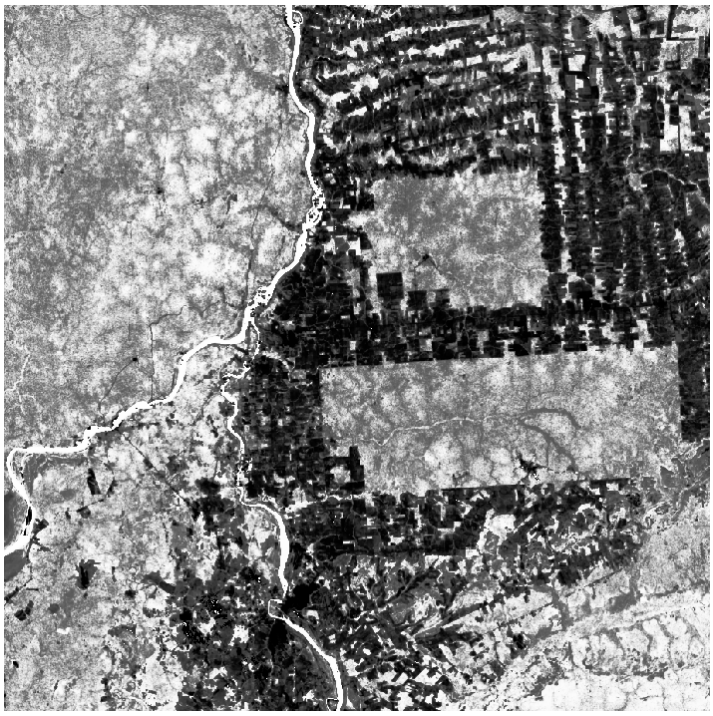


Figure 22: NBR ratio for a regular data cube built using Sentinel-2 tiles and 20LKP and 20LLP

Working with time series

Data structures for satellite time series

The `sits` package requires a set of time series data, describing properties in spatiotemporal locations of interest. For land use classification, this set consists of samples provided by experts that take in-situ field observations or recognize land classes using high-resolution images. The package can also be used for any type of classification, provided that the timeline and bands of the time series (used for training) match that of the data cubes.

For handling time series, the package uses a `tibble` to organize time series data with associated spatial information. A `tibble` is a generalization of a `data.frame`, the usual way in R to organize data in tables. Tibbles are part of the `tidyverse`, a collection of R packages designed to work together in data manipulation [4]. The following code shows the first three lines of a tibble containing 1,882 labeled samples of land cover in Mato Grosso state of Brazil. The samples contain time series extracted from the MODIS MOD13Q1 product from 2000 to 2016, provided every 16 days at 250-meter resolution in the Sinusoidal projection. Based on ground surveys and high-resolution imagery, it includes samples of nine classes: "Forest", "Cerrado", "Pasture", "Soy_Fallow", "Fallow_Cotton", "Soy_Cotton", "Soy_Corn", "Soy_Millet", and "Soy_Sunflower".

```
# data set of samples
data("samples_matogrosso_mod13q1")
samples_matogrosso_mod13q1[1:4, ]
```

```
#> # A tibble: 4 x 7
#>  longitude latitude start_date end_date label cube
#>      <dbl>  <dbl> <date> <date>  <chr> <chr>
#> 1   -57.8  -9.76 2006-09-14 2007-08-29 Pasture bdc_~
#> 2   -59.4  -9.31 2014-09-14 2015-08-29 Pasture bdc_~
#> 3   -59.4  -9.31 2013-09-14 2014-08-29 Pasture bdc_~
#> 4   -57.8  -9.76 2006-09-14 2007-08-29 Pasture bdc_~
#> # ... with 1 more variable: time_series <list>
```

A sits tibble contains data and metadata. The first six columns contain spatial and temporal information, the label assigned to the sample, and the data cube from where the data has been extracted. The first sample has been labeled "Pasture" at location (-58.5631, -13.8844) and is valid for the period (2006-09-14, 2007-08-29). Informing the dates where the label is valid is crucial for correct classification. In this case, the researchers involved in labeling the samples chose to use the agricultural calendar in Brazil. For other applications and other countries, the relevant dates will most likely be different from those used in the example. The `time_series` column contains the time series data for each spatiotemporal location. This data is also organized as a tibble, with a column with the dates and the other columns with the values for each spectral band.

Utilities for handling time series

The package provides functions for data manipulation and displaying information for sits tibble. For example, `sits_labels_summary()` shows the labels of the sample set and their frequencies.

```
sits_labels_summary(samples_matogrosso_mod13q1)
```

```
#> # A tibble: 9 x 3
#>  label      count prop
#>  <chr>      <int> <dbl>
#> 1 Cerrado    379 0.200
#> 2 Fallow_Cotton 29 0.0153
#> 3 Forest     131 0.0692
#> 4 Pasture    344 0.182
#> 5 Soy_Corn   364 0.192
#> 6 Soy_Cotton 352 0.186
#> 7 Soy_Fallow  87 0.0460
#> 8 Soy_Millet 180 0.0951
#> 9 Soy_Sunflower 26 0.0137
```

In many cases, it is helpful to relabel the data set. For example, there may be situations when one wants to use a smaller set of labels, since samples in one label on the original set may not be distinguishable from samples with other labels. We then could use `sits_relabel()`, which requires a conversion list (for details, see `?sits_relabel`).

Given that we have used the tibble data format for the metadata and the embedded time series, one can use the functions from `dplyr`, `tidyr`, and `purrr` packages of the `tidyverse` [4] to process the data. For example, the following code uses `sits_select()` to get a subset of the sample data set with two bands (NDVI and EVI) and then uses the `dplyr::filter()` to select the samples labelled as "Cerrado".

```
# select NDVI band
samples_ndvi <- sits_select(samples_matogrosso_mod13q1,
                      bands = "NDVI")

# select only samples with Cerrado label
samples_cerrado <- dplyr::filter(samples_ndvi,
                           label == "Cerrado")
```

Time series visualisation

Given a small number of samples to display, `plot()` tries to group as many spatial locations together. In the following example, the first 12 samples of "Cerrado" class refer to the same spatial location in consecutive time periods. For this reason, these samples are plotted together.

```
# plot the first 12 samples
plot(samples_cerrado[1:12, ])
```

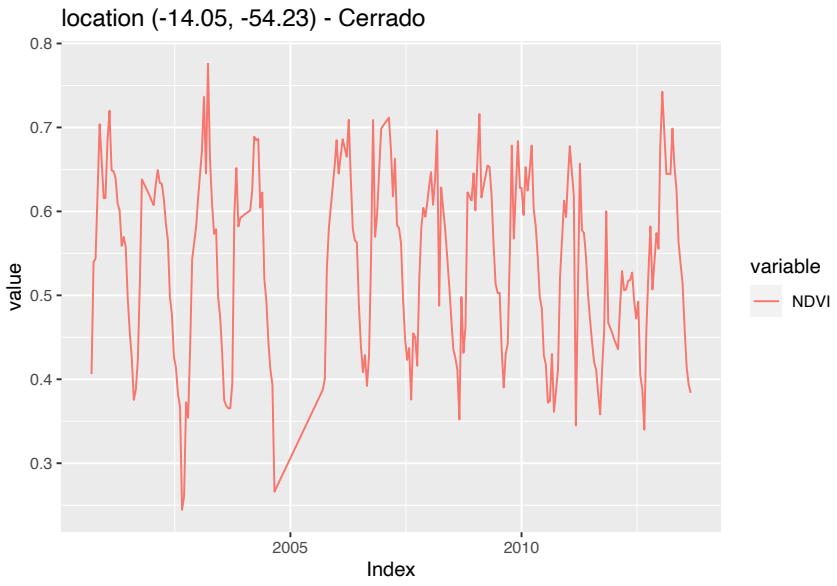


Figure 23: Plot of the first 'Cerrado' sample

For a large number of samples, the default visualization combines all samples together in a single temporal interval even if they belong to different years. This plot is useful to show the spread of values for the time series of each band. The strong red line in the plot shows the median of the values, while the two orange lines are the first and third interquartile ranges. See `?plot` for more details on data visualization in `sits`.

```
# plot all cerrado samples together
plot(samples_cerrado)
```

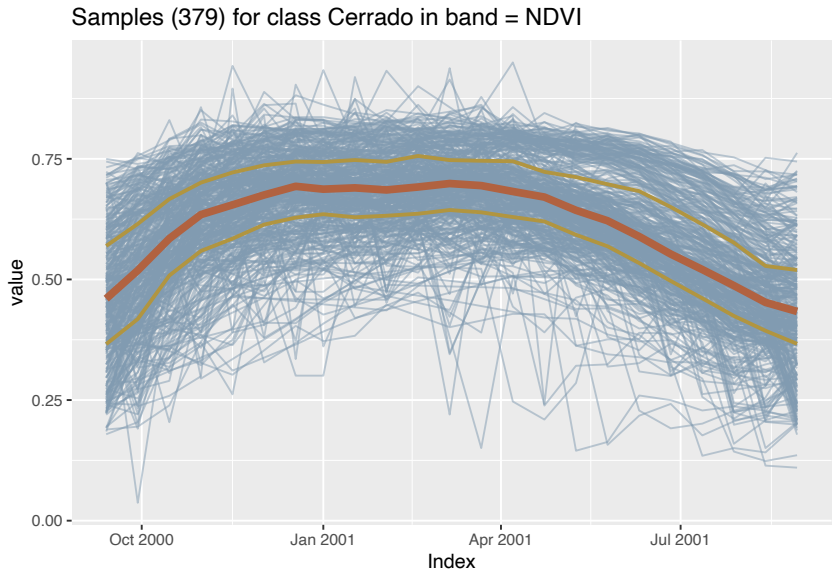


Figure 24: Plot of all Cerrado samples

Obtaining time series data from data cubes

To get a time series in sits, one has to first create a regular data cube and then request one or more time series points from the data cube by using `sits_get_data()`. This function provides a general means of describe training data with the `samples` parameter. This parameter accepts the following data types:

1. A data.frame with information on latitude and longitude (mandatory), start_date, end_date and label for each sample point.
2. A csv file with columns latitude and longitude, start_date, end_date and label.
3. A shapefile containing either POINT or POLYGON geometries. See details below.
4. An `sf` object (from the `sf` package) with POINT or POLYGON geometry information. See details below.

In the example below, given a data cube the user provides the latitude and longitude of the desired location. Since the bands, start date and end date of the time series are not informed, `sits` obtains them from the data cube. The result is a tibble with one time series that can be visualized using `plot()`.

```r
# Obtain a raster cube with based on local files
data_dir <- system.file("extdata/sinop", package = "sitsdata")
raster_cube <- sits_cube(
   source  = "BDC",
   collection = "MOD13Q1-6",
   data_dir = data_dir,
   parse_info = c("X1", "X2", "tile", "band", "date")
)
# obtain a time series from the raster cube from a point
sample_latlong <- tibble::tibble(
   longitude = -55.57320,
   latitude = -11.50566
)
series <- sits_get_data(cube = raster_cube,
                  samples = sample_latlong
)
plot(series)
```

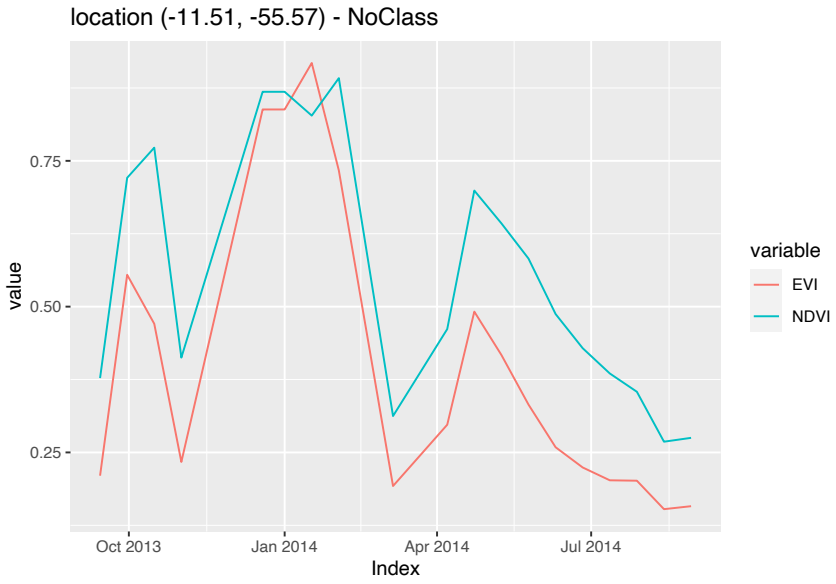


Figure 25: NDVI and EVI time series fetched from local raster cube.

A useful case is when a set of labelled samples are available to be used as a training data set. In this case, one usually has trusted observations that are labelled and commonly stored in plain text files in comma-separated values (CSV) or using shapefiles (SHP).

```
# retrieve a list of samples described by a CSV file
samples_csv_file <- system.file("extdata/samples/samples_sinop_crop.csv",
                        package = "sits")
# for demonstration, read the CSV file into an R object
samples_csv <- read.csv(samples_csv_file)
# print the first three lines
samples_csv[1:3,]
```

```
#> # A tibble: 3 x 6
#>    id longitude latitude start_date end_date label
#>  <int>   <dbl>  <dbl> <chr>  <chr>    <chr>
#> 1   1    -55.7  -11.8 2013-09-14 2014-08-29 Pasture
#> 2   2    -55.6  -11.8 2013-09-14 2014-08-29 Pasture
#> 3   3    -55.7  -11.8 2013-09-14 2014-08-29 Forest
```

To retrieve training samples for time series analysis, users need to provide the temporal information (`start_date` and `end_date`). In the simplest case, all samples share the same dates. That is not a strict requirement. Users can specify different dates, as long as they have a compatible duration. For example, the data set `samples_matogrosso_mod13q1` provided with the `sitsdata` package contains samples from different years covering the same duration. These samples were obtained from the MOD13Q1 product, which contains the same number of images per year. Thus, all time series in the data set `samples_matogrosso_mod13q1` have the same number of instances.

Given a suitably built CSV sample file, `sits_get_data()` requires two parameters: (a) `cube`, the name of the R object that describes the data cube; (b) `samples`, the name of the CSV file.

```
# get the points from a data cube in raster brick format
points <- sits_get_data(
   cube = raster_cube,
   samples = samples_csv_file)
# show the tibble with the first three points
points[1:3,]
```

```
#> # A tibble: 3 x 7
#>  longitude latitude start_date end_date label cube
#>      <dbl>  <dbl> <date> <date>  <chr> <chr>
#> 1   -55.8  -11.7 2013-09-14 2014-08-29 Cerrado MOD1~
#> 2   -55.8  -11.7 2013-09-14 2014-08-29 Cerrado MOD1~
#> 3   -55.7  -11.7 2013-09-14 2014-08-29 Soy_Co~ MOD1~
#> # ... with 1 more variable: time_series <list>
```

Users can also specify samples by providing shapefiles or `sf` objects containing POINT or POLYGON geometries. The geographical location is inferred from the geometries associated with the shapefile or `sf` object. For files containing points, the geographical location is obtained directly. For polygon geometries, the parameter `n_sam_pol` (defaults to 20) determines the number of samples to be extracted from each polygon. The temporal information can be provided explicitly by the user; if absent, it is inferred from the data cube. If label information is available in the shapefile or `sf` object, users should include the parameter `label_attr` to indicate the column which contains the label to be associated with each time series.

```
# obtain a set of points inside the state of Mato Grosso, Brazil
shp_file <- system.file("extdata/shapefiles/mato_grosso/mt.shp",
                  package = "sits")
# read the shapefile into an "sf" object
sf_shape <- sf::st_read(shp_file)
```

```
#> Reading layer 'mt' from data source
#>  '/Library/Frameworks/R.framework/Versions/4.2/Resources/library/sits/
    extdata/shapefiles/mato_grosso/mt.shp'
#>  using driver 'ESRI Shapefile'
#> Simple feature collection with 1 feature and 3 fields
#> Geometry type: POLYGON
#> Dimension: XY
#> Bounding box: xmin: -61.63338 ymin: -18.0416 xmax: -50.22481 ymax:
    -7.349028
#> Geodetic CRS: SIRGAS 2000
```

```
# create a data cube based on MOD13Q1 collection from BDC
modis_cube <- sits_cube(
   source   = "BDC",
   collection = "MOD13Q1-6",
   roi      = sf_shape,
   bands    = c("NDVI", "EVI"),
   start_date = "2020-06-01", end_date = "2021-08-29"
)
# read the points from the cube and produce a tibble with time series
samples_mt <- sits_get_data(
   cube      = modis_cube,
   samples   = shp_file,
   start_date = "2020-06-01", end_date = "2021-08-29",
   n_sam_pol = 20, multicores = 4
)
```

Filtering techniques for time series

Satellite image time series generally is contaminated by atmospheric influence, geolocation error, and directional effects [5]. Atmospheric noise, sun angle, interferences on observations or different equipment specifications, as well as the very nature of the climate-land dynamics can be sources of variability [6]. Inter-annual climate variability also changes the phenological cycles of the vegetation, resulting in time series whose periods and intensities do not match on a year-to-year basis. To make the best use of available satellite data archives, methods for satellite image time series analysis need to deal with *noisy* and *non-homogeneous* data sets. In this vignette, we discuss filtering techniques to improve time series data that present missing values or noise.

The literature on satellite image time series has several applications of filtering to correct or smooth vegetation index data. The package supports the well-known Savitzky–Golay (`sits_sgolay()`) and Whittaker (`sits_whittaker()`) filters. In an evaluation of MERIS NDVI time series filtering for estimating phenological parameters in India, [6] found that the Whittaker filter provides good results. [7] found that the Savitzky-Golay filter is good for reconstruction in tropical evergreen broadleaf forests.

Savitzky–Golay filter

The Savitzky-Golay filter works by fitting a successive array of $2n + 1$ adjacent data points with a d-degree polynomial through linear least squares. The main parameters for the filter are the polynomial degree (d) and the length of the window data points (n). In general, it produces smoother results for a larger value of n and/or a smaller value of d [8]. The optimal value for these two parameters can vary from case to case. In SITS, the user can set the order of the polynomial using the parameter `order` (default = 3), the size of the temporal window with the parameter `length` (default = 5), and the temporal expansion with the parameter `scaling` (default = 1). The following example shows the effect of Savitsky-Golay filter on a point extracted from the MOD13Q1 product, ranging from 2000-02-18 to 2018-01-01.

```
# Take NDVI band of the first sample data set
point_ndvi <- sits_select(point_mt_6bands, band = "NDVI")
# apply Savitzky Golay filter
point_sg <- sits_sgolay(point_ndvi, length = 11)
# merge the point and plot the series
sits_merge(point_sg, point_ndvi) %>%
   plot()
```

Notice that the resulting smoothed curve has both desirable and unwanted properties. For the period 2000 to 2008, the Savitsky-Golay filter removes noise resulting from

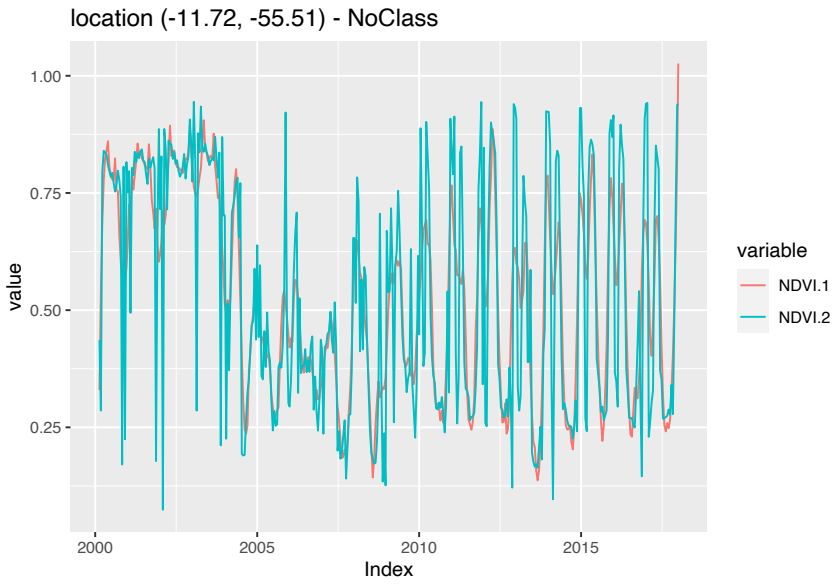


Figure 26: Savitzky-Golay filter applied on a multi-year NDVI time series.

clouds. However, after 2010, when the region has been converted to agriculture, the filter removes an important part of the natural variability from the crop cycle. Therefore, the `length` parameter is arguably too big and results in oversmoothing. Users can try to reduce this parameter and analyse the results.

Whittaker filter

The Whittaker smoother attempts to fit a curve that represents the raw data, but is penalized if subsequent points vary too much [9]. The Whittaker filter is a balancing between the residual to the original data and the "smoothness" of the fitted curve. The filter has one parameter: λ that works as a "smoothing weight" parameter.

The following example shows the effect of Whitakker filter on a point extracted from the MOD13Q1 product, ranging from 2000-02-18 to 2018-01-01. The `lambda` parameter controls the smoothing of the filter. By default, it is set to 0.5, a small value. For illustrative purposes, we show the effect of a larger smoothing parameter.

```
# Take NDVI band of the first sample data set
point_ndvi <- sits_select(point_mt_6bands, band = "NDVI")
# apply Whitakker filter
point_whit <- sits_whittaker(point_ndvi, lambda = 8)
# merge the point and plot the series
sits_merge(point_whit, point_ndvi) %>%
   plot()
```

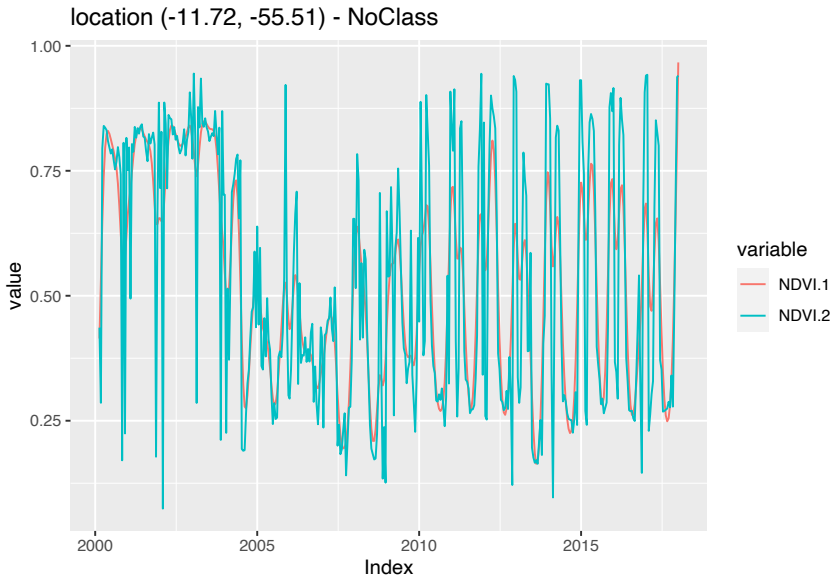


Figure 27: Whittaker filter applied on a one-year NDVI time series.

In the same way as what is observed in the Savitsky-Golay filter, high values of the smoothing parameter `lambda` produce an over-smoothed time series that reduces the capacity of the time series to represent natural variations on crop growth. For this reason, low smoothing values are recommended when using the `sits_whittaker` function.

Improving the Quality of Training Samples

Introduction

One of the key challenges when using samples to train machine learning classification models is assessing their quality. Experience with machine learning methods has shown that the limiting factor in obtaining good results is the number and quality of training samples. Large and accurate data sets are better, no matter the algorithm used [10]; noisy training samples can have a negative effect on classification performance [11]. Therefore, it is useful to apply pre-processing methods to improve the quality of the samples and to remove those that might have been wrongly labeled or that have low discriminatory power.

One needs to distinguish between wrongly labelled samples and differences that result from natural variability of class signatures. When training data is collected over a large geographic region, natural variability of vegetation phenology leads to different patterns being assigned to the same label. A related issue is the limitation of crisp boundaries to describe the natural world. Class definitions use idealized descriptions (e.g., "a savanna woodland has tree cover of 50% to 90% ranging from 8 to 15 meters in height"). In practice, the boundaries between classes are fuzzy and sometimes overlap, making it hard to distinguish between them. Samples quality assessment methods should provide users with means of identifying these different situations.

The package provides support for two clustering methods to test sample quality: (a) Agglomerative Hierarchical Clustering (AHC); (b) Self-organizing Maps (SOM). The two methods have different computational complexities. As discussed below, AHC results are somewhat easier to interpret than those of SOM. However, AHC has a computational complexity of $\mathcal{O}(n^2)$ given the number of time series n, whereas SOM complexity is linear with respect to n. For large data sets, AHC requires an substantial amount of memory and running time; in these cases, SOM is recommended.

Hierachical clustering for sample quality control

Agglomerative hierarchical clustering (AHC) computes the dissimilarity between any two elements from a data set. Depending on the distance functions and linkage criteria, the algorithm decides which two clusters are merged at each iteration. This approach is useful for exploring data samples due to its visualization power and ease of use [12]. In `sits`, AHC is implemented using `sits_cluster_dendro()`.

```
# take a set of patterns for 2 classes
# create a dendrogram, plot, and get the optimal cluster based on ARI
    index
clusters <- sits_cluster_dendro(
   samples = cerrado_2classes,
   bands = c("NDVI", "EVI"),
   dist_method = "dtw_basic",
   linkage = "ward.D2"
)
```

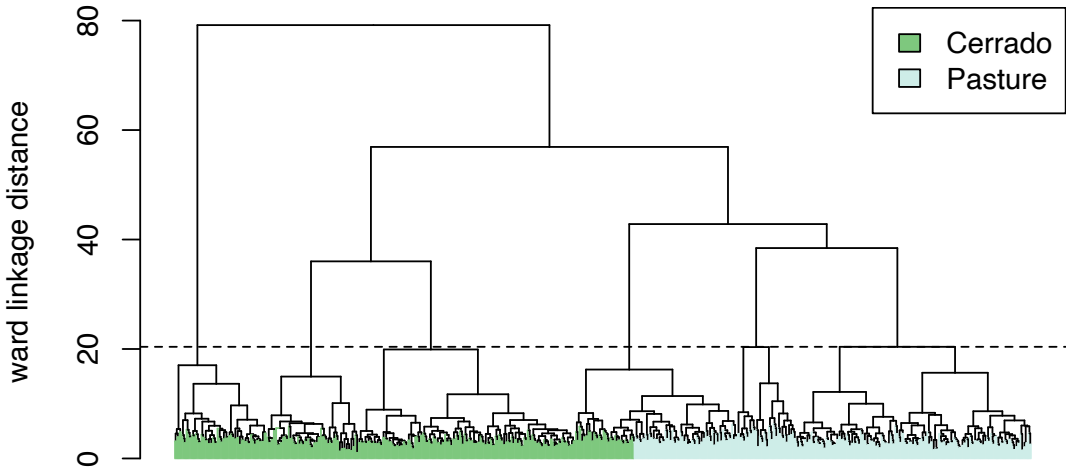


Figure 28: Example of hierarchical clustering for a two class set of time series

The `sits_cluster_dendro()` function has one mandatory parameter (`samples`), where users should provide the name of the R object containing the data samples to be evaluated. Optional parameters include `bands`, `dist_method` and `linkage`. The `dist_method` parameter specifies how to calculate the distance between two time series. We recommend a metric that uses dynamic time warping (DTW)[13], as DTW is reliable method for measuring differences between satellite image time series [14]. The options available in `sits` are based on those provided by package `dtwclust`, which include `dtw_basic`, `dtw_lb`, and `dtw2`. Please check `?dtwclust::tsclust` for more information on DTW distances.

The `linkage` parameter defines the metric used for computing the distance between clusters. The recommended linkage criteria are: `complete` or `ward.D2`. Complete linkage prioritizes the within-cluster dissimilarities, producing clusters with shorter distance samples, but results are sensitive to outliers. As an alternative, Ward proposes to minimize the data variance by means of either *sum-of-squares* or *sum-of-squares-error* [15]. To cut the dendrogram, the `sits_cluster_dendro()` function computes the *adjusted rand index* (ARI) [16] and returns the height where the cut of the dendrogram maximizes the index . In the example, the ARI index indicates that six (6) clusters are present. The result of `sits_cluster_dendro()` is a time series tibble with one additional column, called "cluster". The function `sits_cluster_frequency()` provides information on the composition of each cluster.

```
# show clusters samples frequency
sits_cluster_frequency(clusters)
```

```
#>
#>          1  2  3  4  5  6 Total
#>  Cerrado 203 13 23 80 1 80 400
#>  Pasture 2 176 28 0 140 0  346
#>  Total 205 189 51 80 141 80 746
```

The cluster frequency table shows that each cluster has a predominance of either "Cerrado" or "Pasture" class with the exception of cluster 3 which has a mix of samples from both classes. Such confusion may have resulted from incorrect labeling, inadequacy of selected bands and spatial resolution, or even a natural confusion due to the variability of the land classes. To remove cluster 3 use `dplyr::filter()`. The resulting clusters still contained mixed labels, possibly resulting from outliers. In this case, users may want to remove the outliers and leave only the most frequent class using `sits_cluster_clean()`. After cleaning the samples, the result set of samples is likely to improve the classification results.

```
# remove cluster 3 from the samples
clusters_new <- dplyr::filter(clusters, cluster != 3)
# clear clusters, leaving only the majority class
clean <- sits_cluster_clean(clusters_new)
# show clusters samples frequency
sits_cluster_frequency(clean)
```

```
#>
#>          1  2  4  5  6 Total
#>  Cerrado 203 0 80 0 80  363
#>  Pasture 0 176 0 140 0  316
#>  Total 203 176 80 140 80 679
```

Using self-organizing maps for sample quality control

As an alternative for hierarchical clustering for quality control of training samples, SITS provides a clustering technique based on self-organizing maps (SOM). SOM is a dimensionality reduction technique [17], where high-dimensional data is mapped into a two dimensional map, keeping the topological relations between data patterns. As the shown in the Figure below, the SOM 2D map is composed by units called *neurons*. Each neuron has a weight vector, with the same dimension as the training samples. At the start, neurons are assigned a small random value and then trained by competitive learning. The algorithm computes the distances of each member of the training set to all neurons and finds the neuron closest to the input, called the best matching unit.

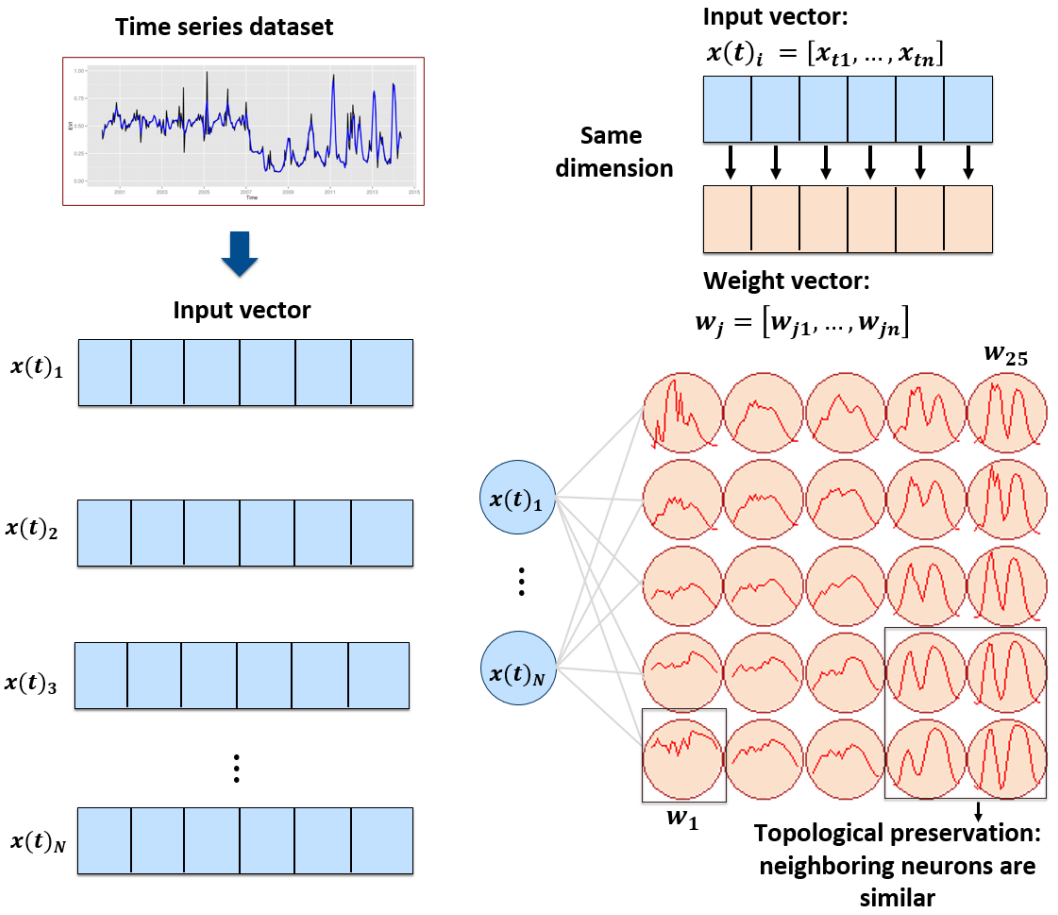


Figure 29: SOM 2D map creation (source: Santos et al.(2021). Reproduction under fair use doctrine.

The input data for quality assessment is a set of training samples, which are high-dimensional data sets such as a time series with 25 instances of 4 spectral bands has 100 dimensions. When projecting a high-dimensional data set into a 2D SOM map,

the units of the map (called *neurons*) compete for each sample. Each time series will be mapped to one of the neurons. Since the number of neurons is smaller than the number of classes, each neuron will be associated to many time series. The resulting 2D map will be a set of clusters. Given that SOM preserves the topological structure of neighborhoods in multiple dimensions, clusters that contain training samples of a given class will usually be neighbors in 2D space. The neighbors of each neuron of a SOM map provide information on intraclass and interclass variability which is used to detect noisy samples. The methodology of using SOM for sample quality assessment (see Figure below) is discussed in detail in the reference paper [18].

As an example, we take a time series dataset from the Cerrado region of Brazil, the second largest biome in South America with an area of more than 2 million km2. This set ranges from 2000 to 2017 and includes 50,160 land use and cover samples divided into 12 classes("Dense_Woodland", "Dunes", "Fallow_Cotton", "Millet_Cotton", "Pasture", "Rocky_Savanna", "Savanna", "Savanna_Parkland", "Silviculture", "Soy_Corn", "Soy_Cotton", "Soy_Fallow"). Each time series covers 12 months (23 data points) from MOD13Q1 product, and has 4 bands ("EVI", "NDVI", "MIR", and "NIR"). We use bands "NDVI" and "EVI" for faster processing.

```
# take only the NDVI and EVI bands
samples_cerrado_mod13q1_2bands <- sits_select(
   data = samples_cerrado_mod13q1,
   bands = c("NDVI", "EVI")
)
# show the summary of the samples
sits_labels_summary(
  data = samples_cerrado_mod13q1_2bands
)
```

```
#> # A tibble: 12 x 3
#>   label        count  prop
#>   <chr>        <int> <dbl>
#> 1 Dense_Woodland 9966 0.199
#> 2 Dunes         550 0.0110
#> 3 Fallow_Cotton 630 0.0126
#> 4 Millet_Cotton 316 0.00630
#> 5 Pasture      7206 0.144
#> 6 Rocky_Savanna 8005 0.160
#> 7 Savanna      9172 0.183
#> 8 Savanna_Parkland 2699 0.0538
#> 9 Silviculture  423 0.00843
#> 10 Soy_Corn    4971 0.0991
#> 11 Soy_Cotton  4124 0.0822
#> 12 Soy_Fallow  2098 0.0418
```

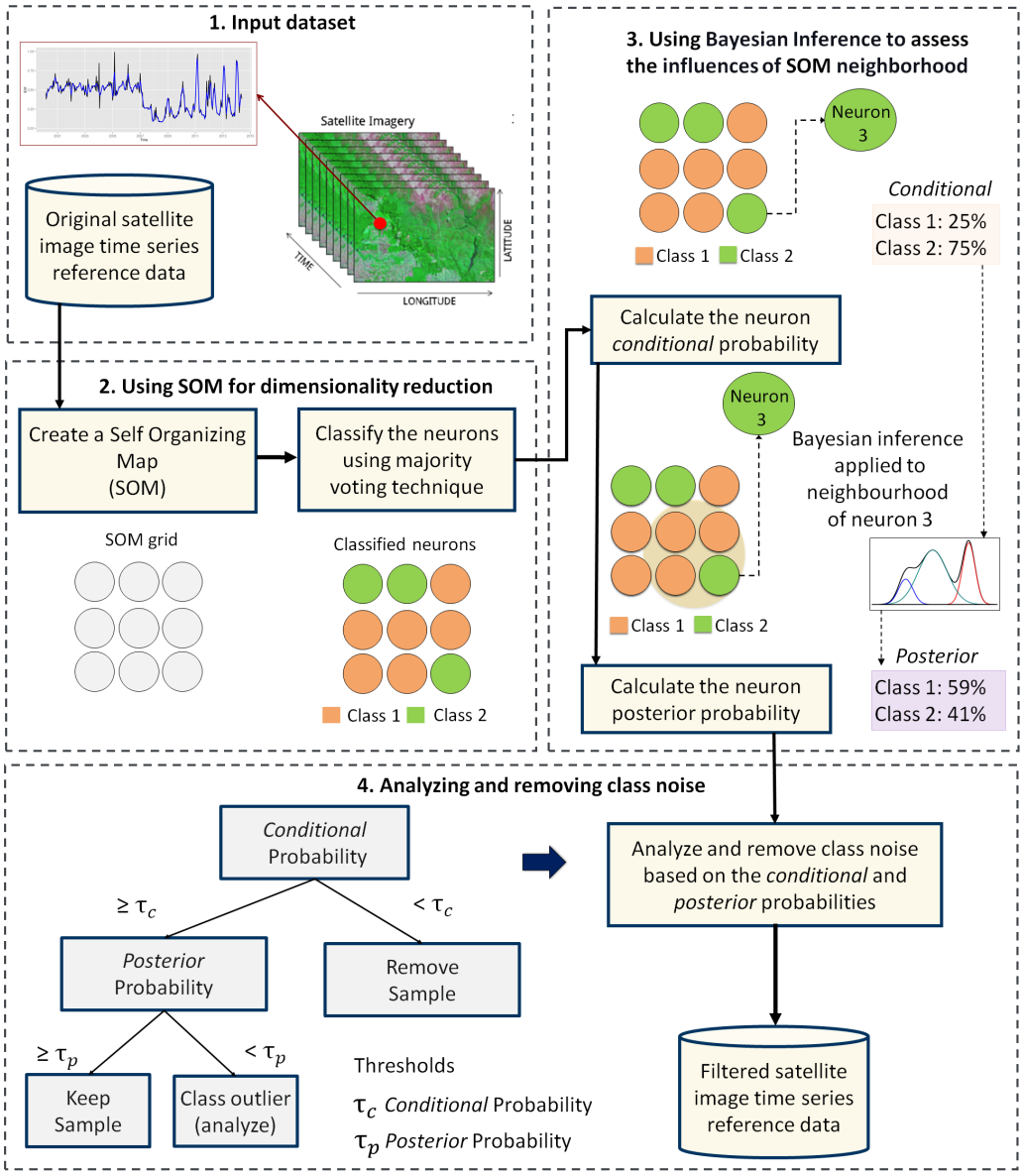


Figure 30: Using SOM for class noise reduction (source: Santos et al.(2021). Reproduction under fair use doctrine.

SOM-based quality assessment: creating the SOM map

To perform the SOM-based quality assessment, the first step is to run `sits_som_map()` which uses the `kohonen` R package [19] to compute a SOM grid, controlled by five parameters. The grid size is given by `grid_xdim` and `grid_ydim`. The starting learning rate is `alpha`, which decreases during the interactions. To measure separation between samples, use `distance` (either "sumofsquares" or "euclidean"). The number of iterations is set by `rlen`. For more details, please consult `?kohonen::supersom`.

```
# clustering time series using SOM
som_cluster <- sits_som_map(samples_cerrado_mod13q1_2bands,
   grid_xdim = 15,
   grid_ydim = 15,
   alpha = 1.0,
   distance = "euclidean",
   rlen = 20
)
# plot the som map
plot(som_cluster)
```

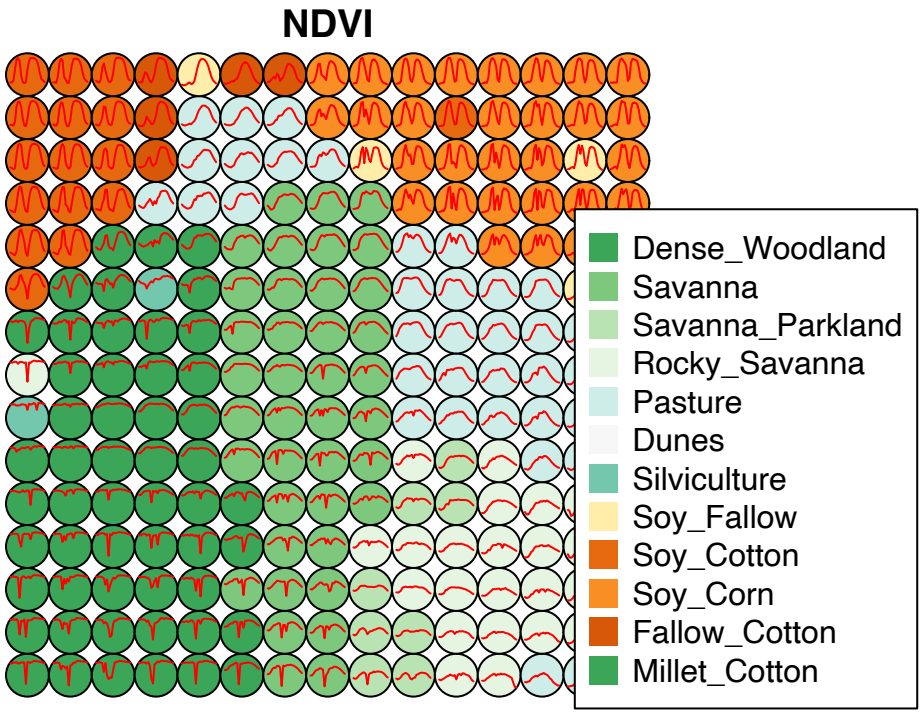


Figure 31: SOM map for the Cerrado samples

The output of the `sits_som_map()` is a list with 3 elements: (a) the original set of time series with two additional columns for each time series: `id_sample` (the original id of each sample) and `id_neuron` (the id of the neuron to which it belongs); (b) a tib-

ble with information on the neurons. For each neuron, it gives the prior and posterior probabilities of all labels which occur in the samples assigned to it; and (c) the SOM grid. To plot the SOM grid, use `plot()`. The neurons are labelled using majority voting.

The SOM grid shows that most classes are associated to neurons close to each other. The are exceptions. Some "Pasture" neurons are far from the main cluster, because the transition between areas of open savanna and pasture is not always well defined and depends on climate and latitude. Also, the neurons associated to "Soy_Fallow" are dispersed in the map; this indicates possible problems in distinguishing this class from the other agricultural classes. Thus, the SOM grid provides a measures of sample quality.

SOM-based quality assessment: measuring confusion between labels

The second step in SOM-based quality assessment is understanding the confusion between labels. The function `sits_som_evaluate_cluster()` groups neurons by their majority label and produces a tibble. For each label, the tibble show the percentage of samples with a different label that have been mapped to a neuron whose majority is that label.

```
# produce a tibble with a summary of the mixed labels
som_eval <- sits_som_evaluate_cluster(som_cluster)
# show the result
som_eval
```

```
#> # A tibble: 79 x 4
#>   id_cluster cluster  class    mixture_percent~
#>        <int> <chr>    <chr>              <dbl>
#> 1          1 Dense_Woodland Dense_Woo~ 81.0
#> 2          1 Dense_Woodland Millet_Co~ 0.00871
#> 3          1 Dense_Woodland Pasture    6.14
#> 4          1 Dense_Woodland Rocky_Sav~ 7.16
#> 5          1 Dense_Woodland Savanna    4.03
#> 6          1 Dense_Woodland Silvicult~ 1.63
#> 7          1 Dense_Woodland Soy_Corn   0.0261
#> 8          1 Dense_Woodland Soy_Cotton 0.0261
#> 9          1 Dense_Woodland Soy_Fallow 0.00871
#> 10         2 Dunes      Dunes         100
#> # ... with 69 more rows
```

As seen above, almost all labels are associated to clusters where there are some samples with a different label. Such confusion between labels arises because visual labeling of samples is subjective and can be biased. In many cases, interpreters use high-resolution data to identify samples. However, the actual images to be classified

are captured by satellites with lower resolution. In our case study, a MOD13Q1 image has pixels with 250 x 250 meter resolution. Therefore, the correspondence between labelled locations in high-resolution images and mid to low-resolution images is not direct. Therefore, the SOM-based analysis is useful to select only homogeneous pixels.

The confusion by class can be visualised in a bar plot using `plot()`, as shown below. The bar plot shows some confusion between the classes associated to the natural vegetation typical of the Brazilian Cerrado ("Savanna", "Savanna_Parkland", "Rocky_Savanna"). This mixture is due to the large variability of the natural vegetation of the Cerrado biome, which makes it difficult to draw sharp boundaries between each label. Some confusion is also visible between the agricultural classes. The "Millet_Cotton" class is a particularly difficult one, since many of the samples assigned to this class are confused with "Soy_Cotton" and "Fallow_Cotton".

```
# plot the confusion between clusters
plot(som_eval)
```

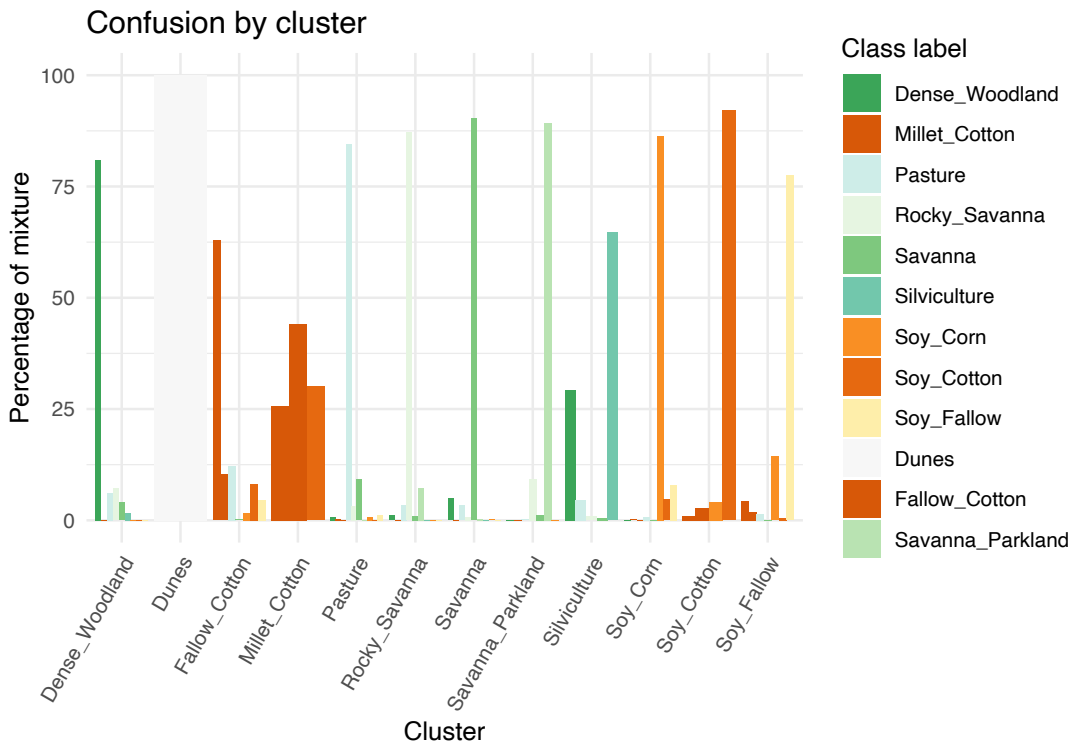


Figure 32: Confusion between classes as measured by SOM.

SOM-based quality assessment part 3: using probabilities to detect noisy samples

The third step in the quality assessment uses the discrete probability distribution associated to each neuron, which is included in the `labelled_neurons` tibble produced by `sits_som_map()`. More homogeneous neurons (those with a single class of high probability) are assumed to be composed of good quality samples. Heterogeneous neurons (those with two or more classes with significant probability) are likely to contain noisy samples. The algorithm computes two values for each sample:

- *prior probability*: the probability that the label assigned to the sample is correct, considering only the samples contained in the same neuron. For example, if a neuron has 20 samples, of which 15 are labeled as "Pasture" and 5 as "Forest", all samples labeled "Forest" are assigned a prior probability of 25%. This is an indication that the "Forest" samples in this neuron may not be of good quality.
- *posterior probability*: the probability that the label assigned to the sample is correct, considering the neighboring neurons. Take the case of the above-mentioned neuron whose samples labeled "Pasture" have a prior probability of 75%. *What happens if all the neighboring samples have "Forest" as a majority label?* To answer this question, we use Bayesian inference to estimate if these samples are noisy based on the neighboring neurons [20].

To identify noisy samples, we take the result of the `sits_som_map()` function as the first argument to the function `sits_som_clean_samples()`. This function finds out which samples are noisy, those that are clean, and some that need to be further examined by the user. It requires the `prior_threshold` and `posterior_threshold` parameters according to the following rules:

- If the prior probability of a sample is less than `prior_threshold`, the sample is assumed to be noisy and tagged as "remove";
- If the prior probability is greater or equal to `prior_threshold` and the posterior probability calculated by Bayesian inference is greater or equal to `posterior_threshold`, the sample is assumed not to be noisy and thus is tagged as "clean";
- If the prior probability is greater or equal to `prior_threshold` and the posterior probability is less than `posterior_threshold`, we have a situation the sample is part of the majority level of those assigned to its neuron, but its label is not consistent with most of its neighbors. This is an anomalous condition and is tagged as "analyze". Users are encouraged to inspect such samples to find out whether they are in fact noisy or not.

The default value for both `prior_threshold` and `posterior_threshold` is 60%. The `sits_som_clean_samples()` has an additional parameter (`keep`) which indicates which samples should be kept in the set based on their prior and posterior probabilities. The default for `keep` is `c("clean", "analyze")`. As a result of the cleaning, about 900 samples have been considered to be noisy and thus removed.

```
new_samples <- sits_som_clean_samples(
   som_map = som_cluster,
   prior_threshold = 0.6,
   posterior_threshold = 0.6,
   keep = c("clean", "analyze")
)
# print the new sample distribution
sits_labels_summary(new_samples)
```

```
#> # A tibble: 11 x 3
#>   label        count  prop
#>   <chr>        <int> <dbl>
#> 1 Dense_Woodland 8214 0.207
#> 2 Dunes         550 0.0139
#> 3 Fallow_Cotton 345 0.00870
#> 4 Pasture      5353 0.135
#> 5 Rocky_Savanna 6289 0.159
#> 6 Savanna      7901 0.199
#> 7 Savanna_Parkland 2106 0.0531
#> 8 Silviculture   85 0.00214
#> 9 Soy_Corn     4191 0.106
#> 10 Soy_Cotton  3489 0.0880
#> 11 Soy_Fallow  1139 0.0287
```

All samples of the class which had the highest confusion with others("Millet_Cotton") have been removed. Most samples of class "Silviculture" (planted forests) have also been removed, since in the SOM map they have been confused with natural forests and woodlands. Further analysis includes calculating the SOM map and confusion matrix for the new set, as shown in the following example.

```
# evaluate the misture in the SOM clusters of new samples
new_cluster <- sits_som_map(
  data = new_samples,
  grid_xdim = 15,
  grid_ydim = 15,
  alpha = 1.0,
  distance = "euclidean",
)
```

```
new_cluster_mixture <- sits_som_evaluate_cluster(new_cluster)
# plot the mixture information.
plot(new_cluster_mixture)
```

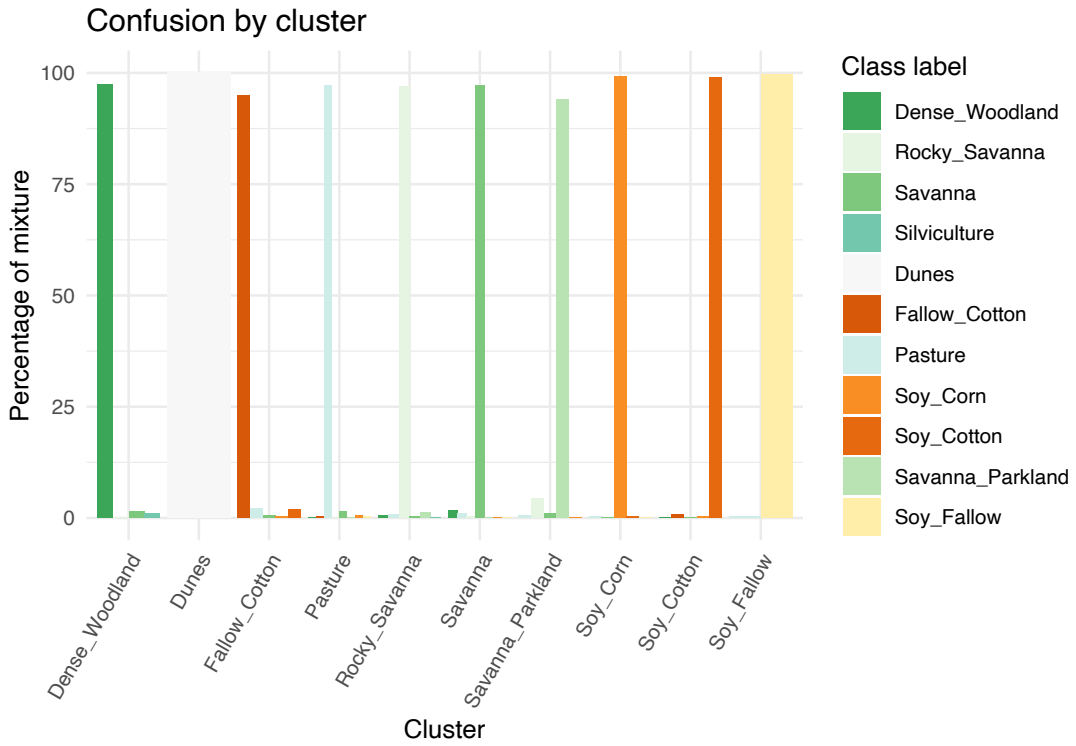


Figure 33: Cluster confusion plot for samples cleaned by SOM

As expected, the new confusion map shows a significant improvement over the previous one. This result should be interpreted carefully, since it may be due to different effects. The most direct interpretation is that "Millet_Cotton" and "Silviculture" cannot be easily separated from the other classes, given the current attributes (a time series of "NDVI" and "EVI" indices from MODIS images). In such situations, users should consider improving the number of samples from the less represented classes, including more MODIS bands, or working with higher resolution satellites. In general, results of the SOM method should be interpreted based on the users' understanding of the ecosystems and agricultural practices of the study region.

A further comparison between the original and clean samples is to run a 5-fold validation on the original and on the cleaned sample sets using `sits_kfold_validate()` and a random forests model. The SOM procedure improves the validation results from 95% on the original data set to 99% in the cleaned one. This improvement should not be interpreted as providing better fit for the final map accuracy. A 5-fold validation procedure only measures how well the machine learning model fits the samples; it is not

an accuracy assessment of classification results. The result indicates that the resulting training set after the SOM sample removal procedure is more internally consistent than the original one. For more details on accuracy measures, please see chapter on "Validation and Accuracy Measures".

```
# run a k-fold validation
assess_orig <- sits_kfold_validate(
   samples = samples_cerrado_mod13q1_2bands,
   folds = 5,
   ml_method = sits_rfor()
)
# print summary
sits_accuracy_summary(assess_orig)
```

```
#> Overall Statistics
#> Accuracy : 0.9457
#>   95% CI : (0.9437, 0.9476)
#>    Kappa : 0.9366
```

```
assess_new <- sits_kfold_validate(
   samples = new_samples,
   folds = 5,
   ml_method = sits_rfor()
)
# print summary
sits_accuracy_summary(assess_new)
```

```
#> Overall Statistics
#> Accuracy : 0.9908
#>   95% CI : (0.9898, 0.9917)
#>    Kappa : 0.9892
```

The SOM-based analysis discards samples which can be confused with samples of other classes. After removing noisy samples or uncertain classes, the data set obtains a better validation score since there is less confusion between classes. Users should analyse the results with care. Not all discarded samples are low quality ones. Confusion between samples of different classes can result from inconsistent labeling or from the lack of capacity of satellite data to distinguish between chosen classes. When many samples are discarded, as in the current example, it is advisable to revise the whole classification schema. The aim of selecting training data should always be to match the reality in the ground to the power of remote sensing data to identify differences. No analysis procedure can replace actual user experience and knowledge of the study region.

Reducing sample imbalance

Many training samples for Earth observation data analysis are imbalanced. This situation arises when the distribution of samples associated to each class is uneven. One example is the Cerrado data set used in this chapter. The three most frequent classes ("Dense Woodland", "Savanna" and "Pasture") include 53% of all samples, while the three least frequent classes ("Millet-Cotton", "Silviculture", and "Dunes") comprise only 2.5% of the data set. Sample imbalance is an undesirable property of a training set. Since machine learning algorithms tend to be more accurate for classes with many samples. The instances belonging to the minority group are misclassified more often than those belonging to the majority group. Thus, reducing sample imbalance can have a positive effect on classification accuracy[21].

The function `sits_reduce_imbalance()` deals with class imbalance; it oversamples minority classes and undersamples majority ones. Oversampling requires generation of synthetic samples. The package uses the SMOTE method that estimates new samples by considering the cluster formed by the nearest neighbors of each minority class. SMOTE takes two samples from this cluster and produces a new one by a random interpolation between them [22].

To perform undersampling for the majority classes, `sits_reduce_imbalance()` builds a SOM map, based on the required number of samples to be selected. Each dimension of the SOM is set to `ceiling(sqrt(new_number_samples/4))` to allow a reasonable number of neurons to group similar samples. After calculating the SOM map, the algorithm extracts four samples per neuron to generate a reduced set of samples that approximates the variation of the original one.

The `sits_reduce_imbalance()` algorithm has two parameters: `n_samples_over` and `n_samples_under`. The first parameter ensures that all classes with samples less than its value are oversampled. The second parameter controls undersampling; all classes with more samples than its value are undersampled. The following example shows the use of `sits_reduce_imbalance()` in the Cerrado data set used in this chapter. We generate a balanced data set where all classes have between 1000 and 1500 samples. We use `sits_som_evaluate_cluster()` to estimate the confusion between classes of the balanced data set.

```
# reducing imbalances in the Cerrado data set
balanced_samples <- sits_reduce_imbalance(
   samples = samples_cerrado_mod13q1_2bands,
   n_samples_over = 1000,
   n_samples_under = 1500,
   multicores = 4
)
```

```
# print the balanced samples
# some classes have more than 1500 samples due to the SOM map
# each class has betwen 10% and 6% of the full set
sits_labels_summary(balanced_samples)
```

```
#> # A tibble: 12 x 3
#>   label        count prop
#>   <chr>        <int> <dbl>
#> 1 Dense_Woodland 1600 0.0966
#> 2 Dunes         1000 0.0604
#> 3 Fallow_Cotton 1000 0.0604
#> 4 Millet_Cotton 1000 0.0604
#> 5 Pasture       1592 0.0961
#> 6 Rocky_Savanna 1520 0.0918
#> 7 Savanna       1600 0.0966
#> 8 Savanna_Parkland 1584 0.0957
#> 9 Silviculture 1000 0.0604
#> 10 Soy_Corn     1592 0.0961
#> 11 Soy_Cotton   1572 0.0949
#> 12 Soy_Fallow   1500 0.0906
```

```
# clustering time series using SOM
som_cluster_bal <- sits_som_map(
   data = balanced_samples,
   grid_xdim = 10,
   grid_ydim = 10,
   alpha = 1.0,
   distance = "euclidean",
   rlen = 20
)
```

```
# produce a tibble with a summary of the mixed labels
som_eval <- sits_som_evaluate_cluster(som_cluster_bal)
```

```
# show the result
plot(som_eval)
```

As shown in the Figure, the balanced data set shows less confusion per class than the unbalanced one. In this case, many of the classes which were confused with other in the original confusion map are now better represented. Reducing class imbalance should be tried as an alternative to reducing the number of samples of the classes using SOM. In general, users should try to balance their training data for better performance.

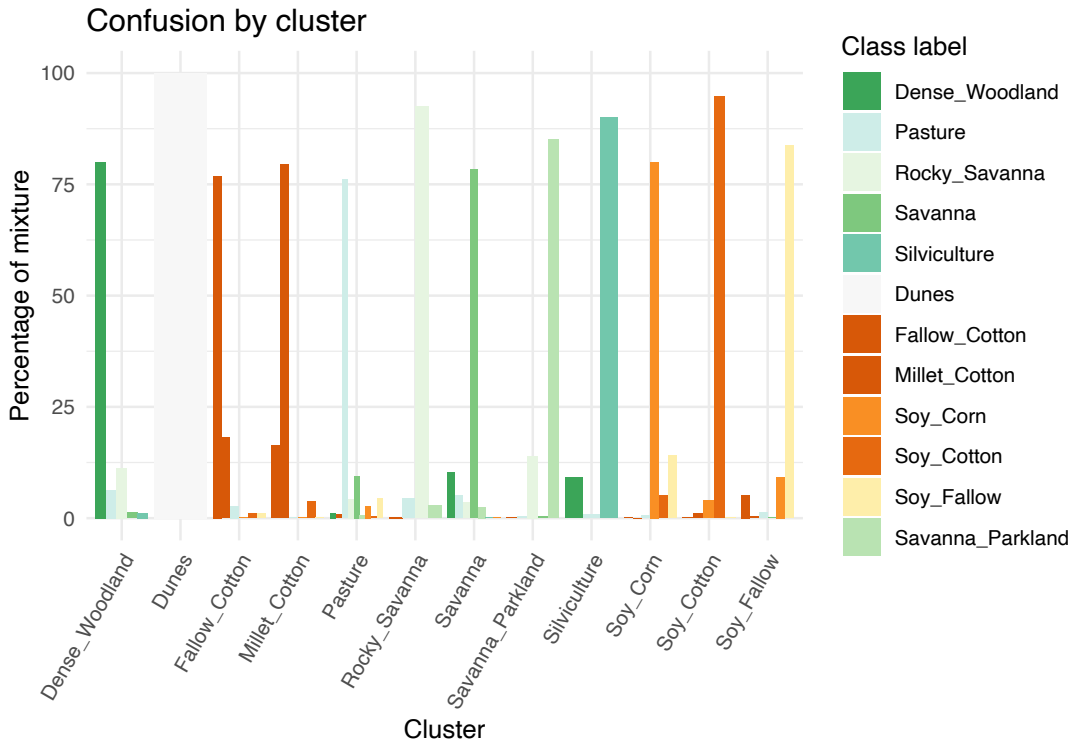


Figure 34: Confusion by cluster for the balanced data set

Conclusion

Machine learning methods are now established as a useful technique for remote sensing image analysis. Despite the well-known fact that the quality of the training data is a key factor in the accuracy of the resulting maps, the literature on methods for detecting and removing class noise in SITS training sets is limited. To contribute to solving this challenge, `sits` provides three methods for training sample improvements. We recommend the use of both imbalance reducing and SOM-based algorithms for large data sets. The SOM-based method identifies potential mislabeled samples and outliers that are flagged to further investigation. The results demonstrate the positive impact on the overall classification accuracy.

We recommend that users dedicate an appropriate time for defining their classification schema. The complexity and diversity of our planet defies simple class names with hard boundaries. Because of representational and data handling issues, all classification systems will have a limited number of categories, which will fail to properly describe the nuances of the planet's landscapes. All representation systems are thus limited and application-dependent. As stated by [23]: *"geographical concepts are situated and context-dependent, can be described from different, equally valid, points of view, and ontological commitments are arbitrary to a large extent"*. The availability of big data and satellite image time series is a further challenge. In principle, image time series can cap-

ture more subtle changes for land classification. In practice, experts need to conceive classification systems and training data collection by understanding how time series information relate to actual land change. Methods for quality analysis such as those presented in this chapter cannot replace actual users understanding and informed choices.

Machine Learning for Data Cubes using the SITS package

Machine learning classification

The `sits` package supports for time series classification, preserving the temporal resolution of the input data. Instead of extracting metrics from time series segments, it uses all values of the time series. Our hypothesis is that building high-dimensional spaces using all values of the time series, coupled with advanced statistical learning methods, is a robust and efficient approach for land cover classification of large data sets. Thus, we use the full depth of satellite image time series to create large dimensional spaces for statistical classification.

The goal of a machine learning models is to approximate a function $y = f(x)$ that maps an input x to a category y. A model defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation [24]. The difference between the different algorithms is the approach they take to build the mapping that classifies the input data.

The following machine learning methods are available in `sits`:

- Random forests (`sits_rfor()`)
- Support vector machines (`sits_svm()`)
- Extreme gradient boosting (`sits_xgboost()`)
- Multilayer perceptrons (`sits_mlp()`)
- Deep Residual Networks (`sits_resnet()`)
- 1D convolutional neural networks (`sits_tempcnn()`)
- Temporal attention encoders (`sits_tae()`)
- Lightweight temporal attention encoders (`sits_lighttae()`)

These methods can be subdivided in two types. The first group does not explicitly consider spatial or temporal dimensions; these algorithms treat time series as a vector in a high-dimensional feature space. They include random forests [25], support vector machines [26], extreme gradient boosting [27], and multilayer perceptrons [28]. The second group of models comprises deep learning methods designed to work with time series. Temporal relations between observed values in a time series are taken into account. From this kind of models, `sits` supports 1D convolution neural networks [29], residual 1D networks [30], and temporal attention-based encoders [31], [32]. In these algorithms, the order of the samples in the time series is relevant for the classifier.

Experience with `sits` shows that SVM, extreme gradient boosting and multilayer perceptron models tend to have inferior performance those using random forests and temporal deep learning. This difference derives from the temporal behavior of land use and land cover classes, where some dates capture more information than others. For example, in deforestation monitoring the dates associates to the actions of forest removal are more informative than earlier or later ones. The same situation occurs in crop mapping, when a large part of the variation is captured in a few dates. In these cases, classification methods which consider the temporal order of samples are more likely to capture the seasonal behavior of the image time series. Random forests methods that use individual measures as nodes in the decision trees can also capture particular events such as deforestation.

The following examples show how to train machine learning methods and apply it to classify a single time series. We use the set `samples_matogrosso_mod13q1`, containing time series samples from Brazilian Mato Grosso state, obtained from the MODIS MOD13Q1 product. It has 1,892 samples and 9 classes (Cerrado, Fallow_Cotton, Forest, Pasture, Soy_Corn, Soy_Cotton, Soy_Fallow, Soy_Millet, Soy_Sunflower). Each time series comprehends 12 months (23 data points) with 6 bands ("NDVI", "EVI", "BLUE", "RED", "NIR", "MIR"). The samples are arranged along an agricultural year, starting in September and ending in August. The data set was used in the paper "Big Earth observation time series analysis for monitoring Brazilian agriculture" [33], and is available in the R package `sitsdata`. Please see the "Setup" section for instructions on how to obtain this pacakge.

The results should not be taken as indication of which method performs better. The most important factor for achieving a good result is the quality of the training data [10]. Experience shows that classification quality depends on the training samples and on how well the model matches these samples. For examples of ML for classifying large areas, please see the "Case Studies" chapter in this book and also the papers by the authors [3], [33]–[35].

Visualizing Sample Patterns

One useful way of describing and understanding the samples is by plotting them. A direct way of doing so is using the `plot` function, as discussed in the "Working with Time Series" chapter. A useful alternative is to estimate a statistical approximation to an idealized pattern based on a generalised additive models (GAM). A GAM is a linear model in which the linear predictor depends linearly on a smooth function of the predictor variables

$$y = \beta_i + f(x) + \epsilon, \epsilon \sim N(0, \sigma^2).$$

The function `sits_patterns()` uses a GAM to predict a smooth, idealized approximation to the time series associated to the each label, for all bands. This function is based on the R package `dtwSat`[36], which implements the TWDTW time series matching method described in [14]. The resulting patterns can be viewed using `plot`.

```
# Estimate the patterns for each class and plot them
samples_matogrosso_mod13q1 %>%
   sits_patterns() %>%
   plot()
```

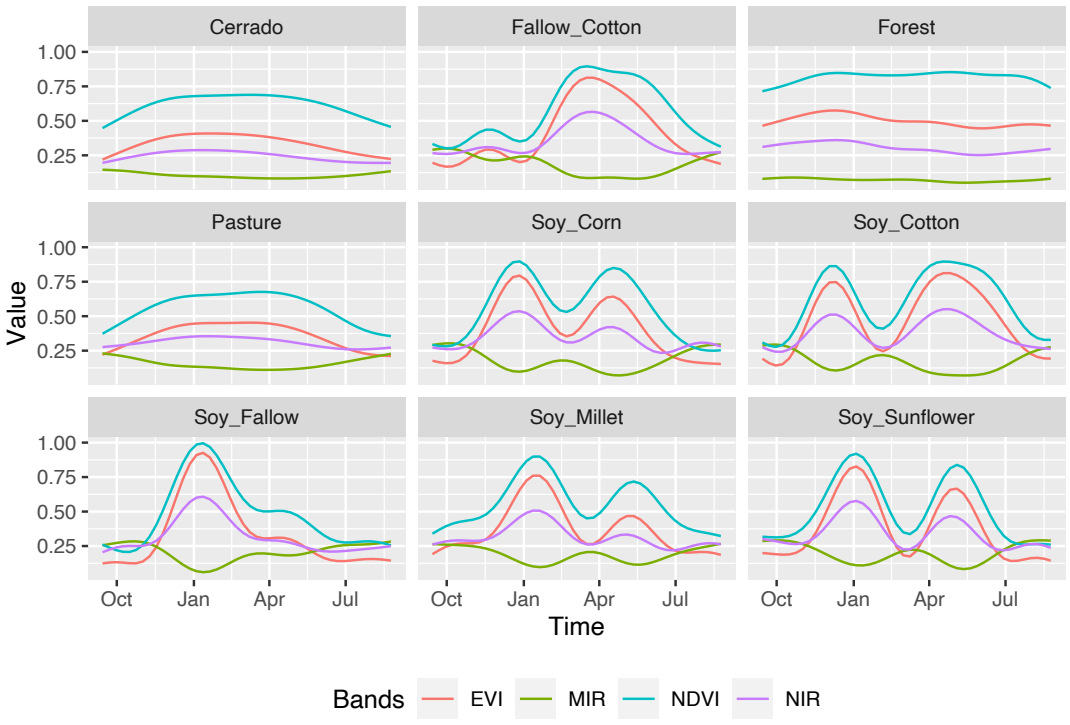


Figure 35: Patterns for the samples for Mato Grosso.

The resulting patterns provide some insights over the time series behavior of each class. The response of the Forest class is quite distinctive. They also shows that it should be possible to separate between the single cropping classes and the double cropping ones. There are similarities between the double-cropping classes (Soy_Corn, Soy_Millet, Soy_Sunflower and Soy_Sunflower) and between the Cerrado and Pasture classes. The subtle differences between class signatures provide hints to possible ways by which machine leaning algorithms might distinguish between classes. One example is the difference between the middle-infrared response during the dry season (May to September) to distinguish between the Cerrado and Pasture classes.

Common interface to machine learning and deep learning models

The SITS package provides a common interface to all machine learning models, using the `sits_train()` function. This function takes two mandatory parameters: the training data (`samples`) and the ML algorithm (`ml_method`). After the model is estimated, it can be used to classify individual time series or data cubes with `sits_classify()`. In what follows, we show how to apply each method for the classification of a single time series. Then, in the "Classification of Images in Data Cubes" we discuss how to classify data cubes.

Since `sits` is aimed at remote sensing users who are not machine learning experts, it provides a set of default values for all classification models. These settings have been chosen based on testing by the authors. Nevertheless, users can control all parameters for each model. Novice users can rely on the default values, while experienced ones can fine-tune model parameters to meet their needs. Model tuning is discussed at the end of this chapter.

When a set of time series organised as tibble is taken as input to the classifier, the result is the same tibble with one additional column ("predicted"), which contains the information on what labels are have been assigned for each interval. The results can be shown in text format using the function `sits_show_prediction()` or graphically using `plot()`.

Random forests

The random forests model uses *decision trees* as its base model with refinements. When building the decision trees, each time a split in a tree is considered, a random sample of `m` features is chosen as split candidates from the full set of `n` features of the samples[37]. Each of these features is then tested; the one maximizing the decrease in a purity mea-

sure is used to build the trees. This criterion is used to identify relevant features and to perform variable selection. This decreases the correlation among trees and improves prediction performance. Classification performance depends on the number of trees in the forest as well as the number of features randomly selected at each node.

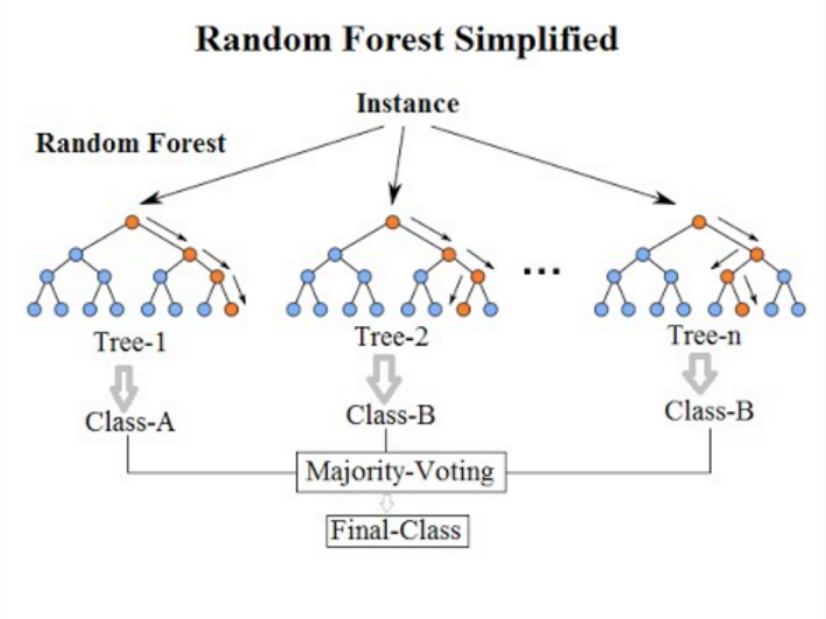


Figure 36: Random forests algorithm (source: Venkata Jagannath in Wikipedia - licenced as CC-BY-SA 4.0.)

SITS provides `sits_rfor()`, which uses the R `randomForest` package [38]; its main parameter is `num_trees`, which is the number of trees to grow with a default value 100. The model can be visualized using `plot()`.

```
# Train the Mato Grosso samples with Random Forests model.
rfor_model <- sits_train(
   samples = samples_matogrosso_mod13q1,
   ml_method = sits_rfor(num_trees = 100)
)
# plot the most important variables of the model
plot(rfor_model)
```

The most important explanatory variables are the NIR (near infrared) band on date 17 (2007-05-25) and the MIR (middle infrared) band in date 22 (2007-08-13). The NIR value at the end of May captures the growth of the second crop for double cropping classes. Values of the MIR band at the end of the period (late July to late August) capture bare soil signatures to distinguish between agricultural classes and natural ones. This corresponds to summer time when the ground is drier and crops have been harvested.

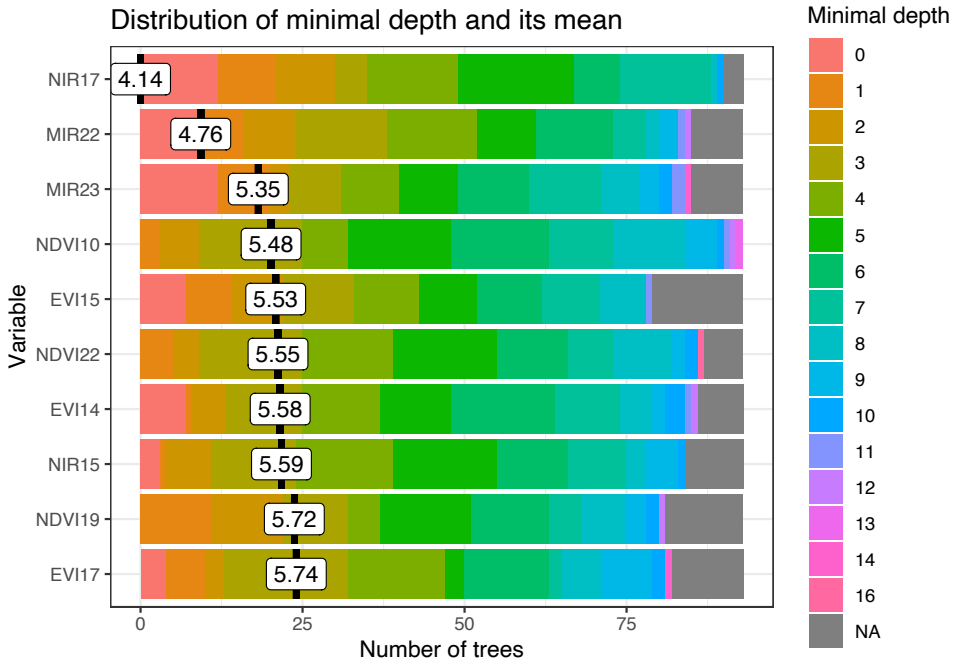


Figure 37: Most important variables in random forests model.

```
# retrieve a point to be classified
point_mt_4bands <- sits_select(
    data = point_mt_6bands,
    bands = c("NDVI", "EVI", "NIR", "MIR")
)
# Classify using Random Forest model and plot the result
point_class <- sits_classify(
    data = point_mt_4bands,
    ml_model = rfor_model
)
plot(point_class, bands = c("NDVI", "EVI"))
```

The result shows that the area started out as a forest in 2000, it was deforested from 2004 to 2005, used as pasture from 2006 to 2007, and for double-cropping agriculture from 2009 onwards. They are consistent with expert evaluation of the process of land use change in this region of Amazonia.

Random forests are robust to outliers and able to deal with irrelevant inputs [39]. The method tends to overemphasize some variables because its performance tends to stabilize after a part of the trees are grown [39]. In cases where abrupt change takes place, such deforestation mapping, random forests (if properly trained) will emphasize the temporal instances and bands that capture such quick change.

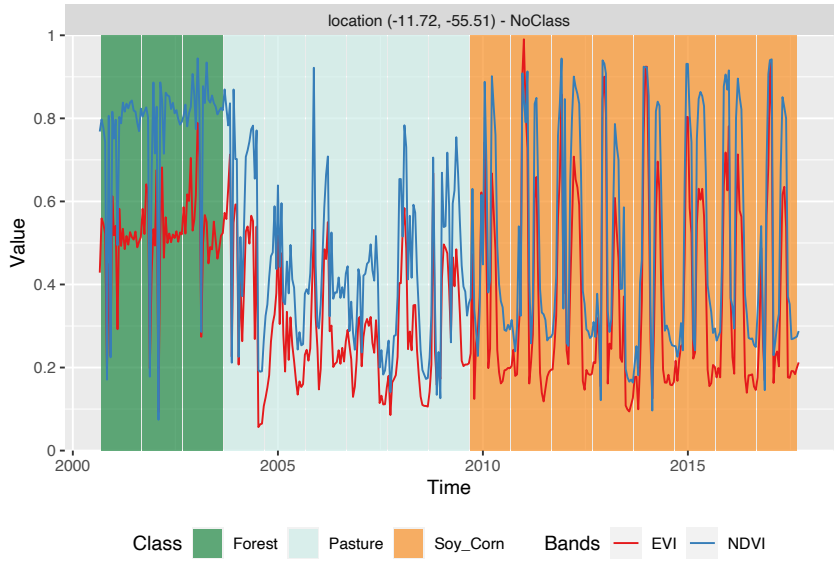


Figure 38: Classification of time series using random forests.

Support Vector Machines

The support vector machine (SVM) classifier is a generalization of a linear classifier which finds an optimal separation hyperplane that minimizes misclassification [40]. Since a set of samples with n features defines an n-dimensional feature space, hyperplanes are linear $(n-1)$-dimensional boundaries that define linear partitions in that space. If the classes are linearly separable on the feature space, there will be an optimal solution defined by the *maximal margin hyperplane*, which is the separating hyperplane that is farthest from the training observations[37]. The maximal margin is computed as the the smallest distance from the observations to the hyperplane. The solution for the hyperplane coefficients depends only on the samples that define the maximum margin criteria, the so-called *support vectors*.

For data that is not linearly separable, SVM includes kernel functions that map the original feature space into a higher dimensional space, providing nonlinear boundaries to the original feature space. The new classification model, despite having a linear boundary on the enlarged feature space, generally translates its hyperplane to a nonlinear boundaries in the original attribute space. Kernels are an efficient computational strategy to produce nonlinear boundaries in the input attribute space; thus, they improve training-class separation. SVM is one of the most widely used algorithms in machine learning applications and has been applied to classify remote sensing data [26].

In `sits`, SVM is implemented as a wrapper of `e1071` R package that uses the `LIBSVM` implementation [41], the `sits` package adopts the *one-against-one* method for multiclass classification. For a q class problem, this method creates $q(q-1)/2$ SVM binary

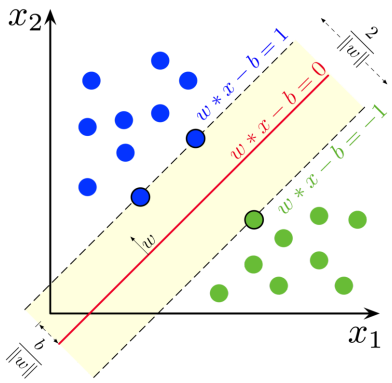


Figure 39: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors. (source: Larhmam in Wikipedia - licensed as CC-BY-SA-4.0).

models, one for each class pair combination and tests any unknown input vectors throughout all those models. The overall result is computed by a voting scheme.

The example below shows how to apply the SVM method for classification of time series using default values. The main parameters for the SVM are `kernel` which controls whether to use a non-linear transformation (default is `radial`), `cost` which measures the punishment for wrongly-classified samples (default is 10), and `cross` which sets the value of the k-fold cross validation (default is 10).

```
# Train a machine learning model for the mato grosso dataset using SVM
svm_model <- sits_train(
   samples = samples_matogrosso_mod13q1,
   ml_method = sits_svm())
# Classify using SVM model and plot the result
point_class <- sits_classify(
   data = point_mt_4bands,
   ml_model = svm_model
)
# plot the result
plot(point_class, bands = c("NDVI", "EVI"))
```

The SVM classifier is less stable and less robust to outliers than the random forests method. In this example, it tends to misclassify some of the data. In 2008, it is likely that the land cover was still "Pasture" rather than "Soy_Millet" as produced by the algorithm, while the "Soy_Cotton" class in 2012 is also inconsistent with the previous and latter classification of "Soy_Corn".

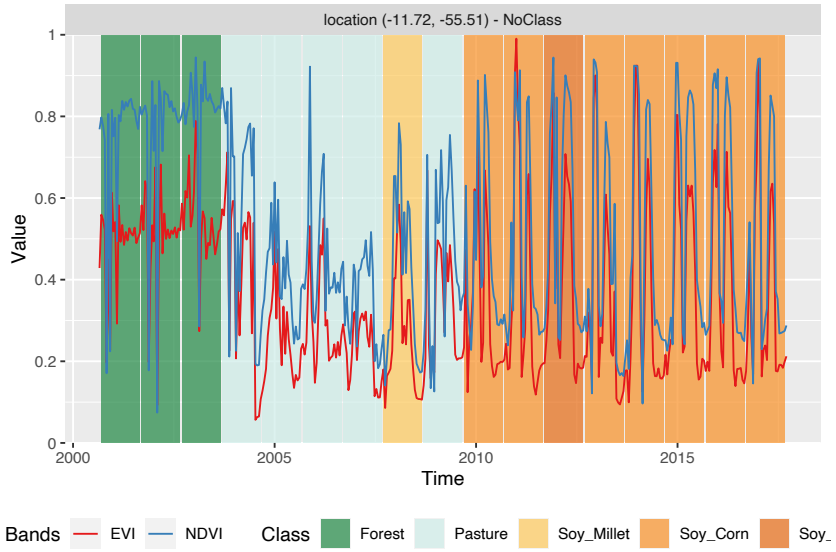


Figure 40: Classification of time series using SVM.

Extreme Gradient Boosting

The boosting method is based on the idea of starting from a weak predictor and then improving performance sequentially by fitting a better model at each iteration. It starts by fitting a simple classifier to the training data, and using the residuals of the fit to build a predictor. Typically, the base classifier is a regression tree. Although both random forests and boosting use trees for classification, there are important differences. The performance of random forests generally increases with the number of trees until it becomes stable. Boosting trees improve on previous result by applying finer divisions that improve the performance [39]. However, the number of trees grown by boosting techniques has to be limited at the risk of overfitting.

Gradient boosting is a variant of boosting methods where the cost function is minimized by gradient descent. Extreme gradient boosting [27], called XGBoost, is an efficient approximation to the gradient loss function. Some recent papers show that it outperforms random forests for remote sensing image classification**Jafarzadeh2021?**. However, this result is not generalizable, since actual performance is controlled by the quality of the training data set.

In SITS, the XGBoost method is implemented by the `sits_xbgoost()` function, which is based on `XGBoost` R package and has five hyperparameters that require tuning. The `sits_xbgoost()` function takes the user choices as input to a cross validation to determine suitable values for the predictor.

The learning rate `eta` varies from 0.0 to 1.0 and should be kept small (default is 0.3) to avoid overfitting. The minimum loss value `gamma` specifies the minimum reduction required to make a split. Its default is 0; increasing it makes the algorithm more con-

servative. The `max_depth` value controls the maximum depth of the trees. Increasing this value will make the model more complex and more likely to overfit (default is 6). The `subsample` parameter controls the percentage of samples supplied to a tree. Its default is 1 (maximum). Setting it to lower values means that xgboost randomly collects only part of the data instances to grow trees, thus preventing overfitting. The `nrounds` parameter controls the maximum number of boosting interactions; its default is 100, which has proven to be enough in most cases. In order to follow the convergence of the algorithm, users can turn the `verbose` parameter on. In general, the results using the extreme gradient boosting algorithm are similar to the random forests method.

```
# Train a model for the mato grosso dataset using XGBOOST
# The default parameters are those of the "xgboost" package
xgb_model <- sits_train(
   samples = samples_matogrosso_mod13q1,
   ml_method = sits_xgboost(verbose = 0)
)
# Classify using SVM model and plot the result
point_class <- sits_classify(
   data = point_mt_4bands,
   ml_model = xgb_model
)
plot(point_class, bands = c("NDVI", "EVI"))
```

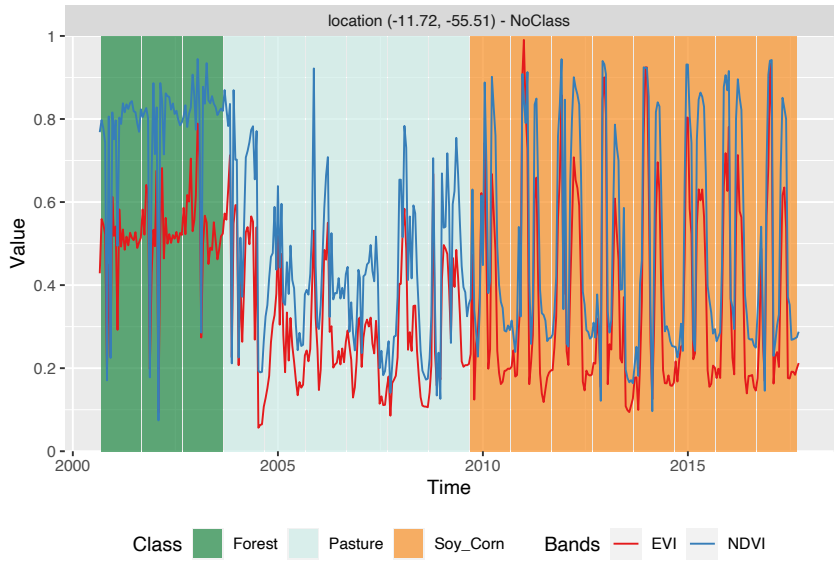


Figure 41: Classification of time series using XGBoost.

Deep learning using multilayer perceptrons

To support deep learning methods, sits uses the torch R package, which takes the Facebook torch C++ library as a back-end. Machine learning algorithms that use the R torch package are similar from those developed using PyTorch. Converting image time series algorithms developed in PyTorch to be using in sits is straightforward. Please see the chapter on "Extensibility" for guidance on how to include new deep learning algorithms in sits.

The simplest deep learning method is multilayer perceptrons (MLPs), which are feedforward artificial neural networks. A MLP consists of three kinds of nodes: an input layer, a set of hidden layers and an output layer. The input layer has the same dimension as the number of the features in the data set. The hidden layers attempt to approximate the best classification function. The output layer makes a decision about which class should be assigned to the input.

In sits, users build MLP models using sits_mlp(). Since there is no established model for generic classification of satellite image time series, designing MLP models requires parameter customization. The most important decisions are the number of layers in the model and the number of neurons per layer. These values are set by the layers parameters, which is a list of integer values. The size of the list is the number of layers and each element of the list indicates the number of nodes per layer.

The choice of the number of layers depends on the inherent separability of the data set to be classified. For data sets where the classes have different signatures, a shallow model (with 3 layers) may provide appropriate responses. More complex situations require models of deeper hierarchy. The user should be aware that some models with many hidden layers may take a long time to train and may not be able to converge. The suggestion is to start with 3 layers and test different options of number of neurons per layer, before increasing the number of layers.

MLP models also need to include the activation function. The activation function of a node defines the output of that node given an input or set of inputs. Following standard practices [24], we use the relu activation function.

Users can also define the optimization method (optimizer), which defines the gradient descent algorithm to be used. These methods aim to maximize an objective function by updating the parameters in the opposite direction of the gradient of the objective function [42]. Based on experience with image time series, we recommend that users start by using the default method provided by sits, which is the optimizer_adamw method from package torchopt. Please refer to the torchopt package for additional information.

Another relevant parameter is the list of dropout rates (dropout). Dropout is a technique for randomly dropping units from the neural network during training [43]. By

randomly discarding some neurons, dropout reduces overfitting. Since the purpose of a cascade of neural nets is to improve learning as more data is acquired, discarding some neurons may seem a waste of resources. In practice, dropout prevents an early convergence to a local minimum [24]. We suggest users experiment with different dropout rates, starting from small values (10-30%) and increasing as required.

The following example shows how to use `sits_mlp()`. The default parameters for have been chosen based on a modified verion of [44], which proposes the use of multilayer perceptrons as a baseline for time series classification. These parameters are: (a) Three layers with 512 neurons each, specified by the parameter `layers`; (b) Using the "relu" activation function; (c) dropout rates of 40%, 30%, and 20% for the layers; (d) the "optimizer_adamw" as optimizer (default value); (e) a number of training steps (`epochs`) of 100; (f) a `batch_size` of 64, which indicates how many time series are used for input at a given steps; and (g) a validation percentage of 20%, which means 20% of the samples will be randomly set side for validation.

In our experience, if the training dataset is of good quality, using 3 to 5 layers is a reasonable compromise. Further increase on the number of layers will not improve the model. To simplify the output generation, the `verbose` option has been turned off. The default value is "on". After the model has been generated, we plot its training history. In this and in the following examples of using deep learning classifiers, both the training samples and the point to be classified are filtered with `sits_whittaker()` with a small smoothing parameter (lambda = 0.5). Since deep learning classifiers are not as robust as Random Forest or XGBoost, the right amount of smoothing improves their detection power in case of noisy data.

```
# train a model for the Mato Grosso data using an MLP model
# this is an example of how to set parameters
# first-time users should test default options first
mlp_model <- sits_train(
   samples = samples_matogrosso_mod13q1,
   ml_method = sits_mlp(
      layers       = c(512, 512, 512),
      dropout_rates = c(0.40, 0.30, 0.20),
      epochs       = 100,
      batch_size   = 64,
      verbose      = FALSE,
      validation_split = 0.2
   )
)
# show training evolution
plot(mlp_model)
```

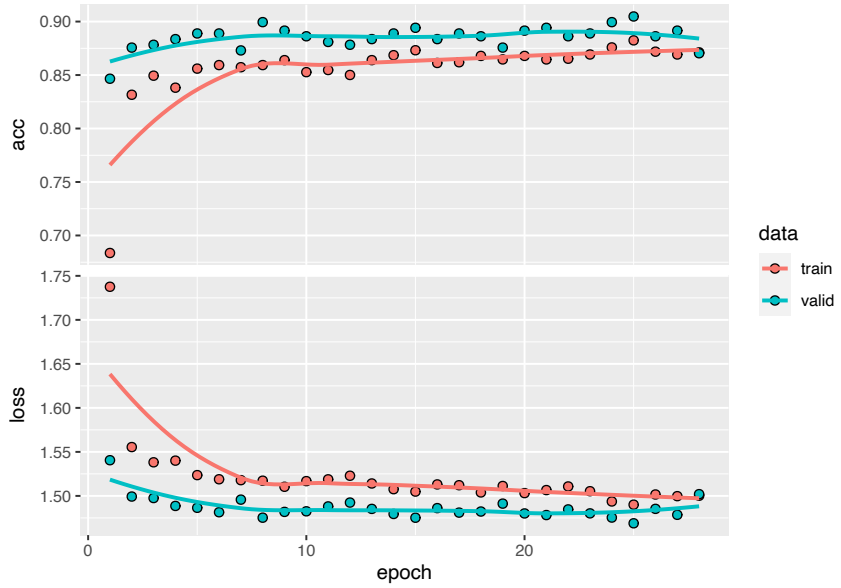


Figure 42: Evolution of training accuracy of MLP model.

Then, we classify a 16-year time series using the multilayer perceptron model.

```
# Classify using DL model and plot the result
point_mt_6bands %>%
   sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
   sits_classify(mlp_model) %>%
   plot(bands = c("NDVI", "EVI"))
```

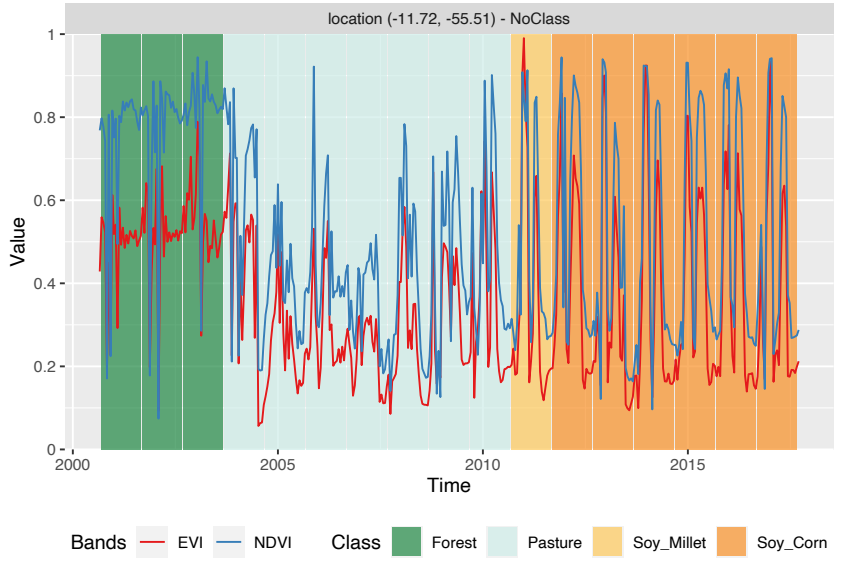


Figure 43: Classification of time series using MLP.

In theory, multilayer perceptron model is able to capture more subtle changes than the random forests and XGBoost models. In this specific case, the result is similar to the them. Although the model mixes the "Soy_Corn" and "Soy_Millet" classes, the distinction between their temporal signatures is quite subtle. Also in this case, this suggests the need to improve the number of samples. In this examples, the MLP model shows an increase in the sensitivity compared to previous models. We recommend that the users compare different configurations, since the MLP model is sensitive to changes in its parameters.

Temporal Convolutional Neural Network (TempCNN)

Convolutional neural networks (CNN) are a variety of deep learning methods where a convolution filter (sliding window) is applied to the input data. In the case of time series, a 1D CNN works by applying a moving window to the series. Using convolution filters is a way to incorporate temporal autocorrelation information in the classification. The result of the convolution is another time series. Russwurm and Körner [45] state that the use of 1D-CNN for time series classification improves on the use of multilayer perceptrons, since the classifier is able to represent temporal relationships; also, the convolution window makes the classifier more robust to moderate noise, e.g. intermittent presence of clouds.

The use of 1D CNNs for satellite image time series classification is proposed by Pelletier et al [29]. The TempCNN architecture has three 1D convolutional layers, and a final softmax layer for classification (see figure). The authors use a combination of different methods to avoid overfitting and reduce the vanishing gradient effect, including dropout, regularization, and batch normalisation. In the tempCNN paper [29], the authors compare favorably their model with the Recurrent Neural Network proposed by Russwurm and Körner [46] for land use classification. The figure below shows the architecture of the tempCNN model.

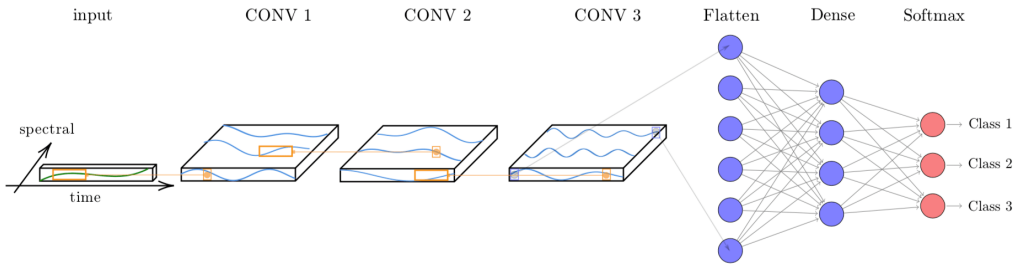


Figure 44: Structure of tempCNN architecture (source: Pelletier et al.(2019))

The function `sits_tempcnn()` implements the model. The parameter `cnn_layers` controls the number of 1D-CNN layers and the size of the filters applied at each layer;

the default values are three CNNs with 128 units. The parameter `cnn_kernels` indicates the size of the convolution kernels; the default are kernels of size 7. Activation for all 1D-CNN layers uses the "relu" function. The dropout rates for each 1D-CNN layer are controlled individually by the parameter `cnn_dropout_rates`. The `validation_split` controls the size of the test set, relative to the full data set. We recommend to set aside at least 20% of the samples for validation.

```
library(torchopt)
# train a machine learning model using tempCNN
tempcnn_model <- sits_train(samples_matogrosso_mod13q1,
                  sits_tempcnn(
                    optimizer       = torchopt::optim_adamw,
                    cnn_layers      = c(128, 128, 128),
                    cnn_kernels     = c(7, 7, 7),
                    cnn_dropout_rates = c(0.2, 0.2, 0.2),
                    epochs          = 100,
                    batch_size      = 64,
                    validation_split = 0.2,
                    verbose         = FALSE) )

# show training evolution
plot(tempcnn_model)
```

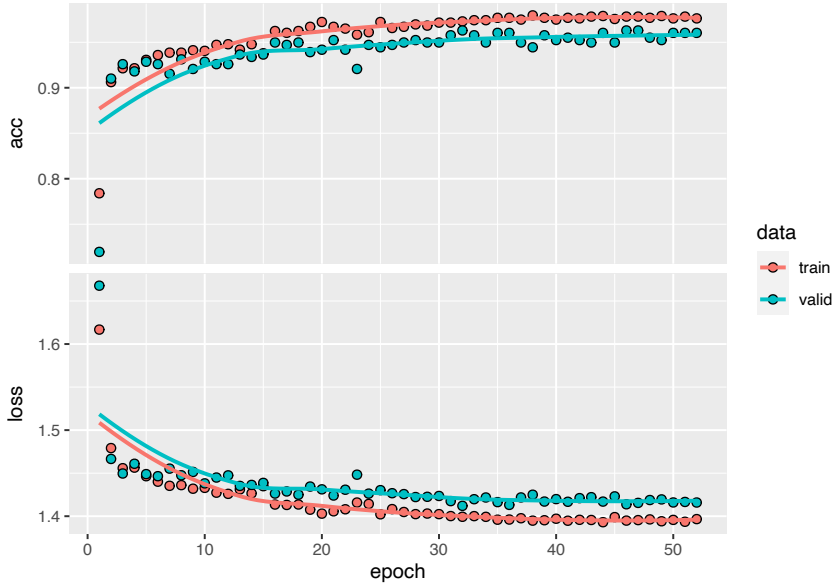


Figure 45: Training evolution of TempCNN model.

Then, we classify a 16-year time series using the TempCNN model.

```
# Classify using TempCNN model and plot the result
class <- point_mt_6bands %>%
```

```
  sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
  sits_classify(tempcnn_model) %>%
  plot(bands = c("NDVI", "EVI"))
```

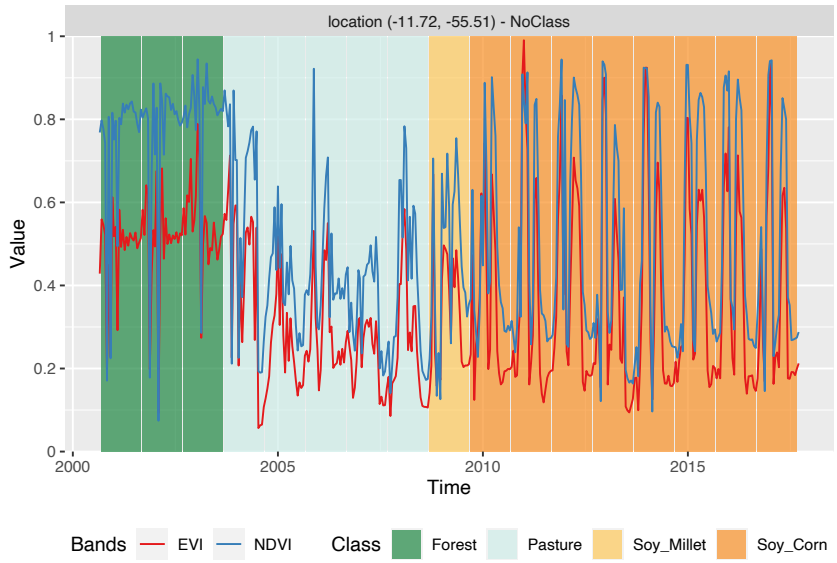


Figure 46: Classification of time series using TempCNN.

The result has important differences from the previous ones. The TempCNN model indicates the "Soy_Cotton" class as the most likely one in 2004. While this result is likely to be wrong, it shows that the time series for year 2004 is different from those of "Forest" and "Pasture" classes. One possible explanation is that there was forest degradation in 2004, leading to a signature that is a mix of forest and bare soil. In this case, users could consider improving the training data by including samples that represent forest degradation. In our experience, TempCNN models are a reliable way for classifying image time series [47]. Recent work which compares different models also provides evidence of that TempCNN models have satisfactory behavior, especially in the case of crop classes [32].

Residual 1D CNN Networks (ResNet)

The Residual Network (ResNet) is a 1D convolution neural network (CNN) proposed by Wang et al. [44], based on the idea of deep residual networks for 2D image recognition [48]. The ResNet architecture is composed of 11 layers, with three blocks of three 1D CNN layers each (see figure below). Each block corresponds to a 1D CNN architecture. The output of each block is combined with a shortcut that links its output to its input, called a "skip connection". The purpose of combining the input layer of each block with its output layer (after the convolutions) is to avoid the so-called "vanishing

gradient problem". This issue occurs in deep networks as he neural network's weights are updated based on the partial derivative of the error function. If the gradient is too small, the weights will not be updated, stopping the training[49]. Skip connections aim to avoid vanishing gradients from occurring, allowing deep networks to be trained.

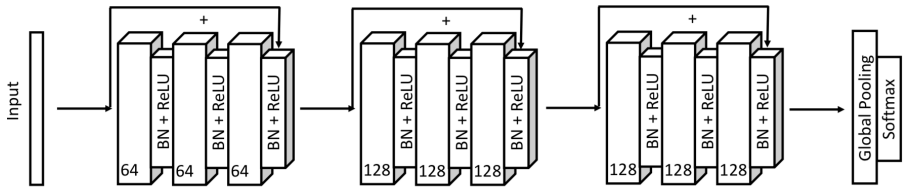


Figure 47: Structure of ResNet architecture (source: Wang et al.(2017)).

In `sits`, the Residual Network is implemented using the `sits_resnet()` function. The default parameters are those proposed by Wang et al. [44], as implemented by Fawaz et al.[50]. The first parameter is `blocks`, which controls the number of blocks and the size of filters in each block. By default, the model implements three blocks, the first with 64 filters and the others with 128 filters. Users can control the number of blocks and filter size by changing this parameter. The parameter `kernels` controls the size the of kernels of the three layers inside each block. We have found out that it is useful to experiment a bit with these kernel sizes in the case of satellite image time series. The default activation is "relu", which is recommended in the literature to reduce the problem of vanishing gradients. The default optimizer is `optim_adamw`, available in package `torchopt`.

```
# train a machine learning model using ResNet
resnet_model <- sits_train(samples_matogrosso_mod13q1,
                  sits_resnet(
                    blocks          = c(64, 128, 128),
                    kernels         = c(7, 5, 3),
                    epochs          = 100,
                    batch_size      = 64,
                    validation_split = 0.2,
                    verbose         = FALSE) )
# show training evolution
plot(resnet_model)
```

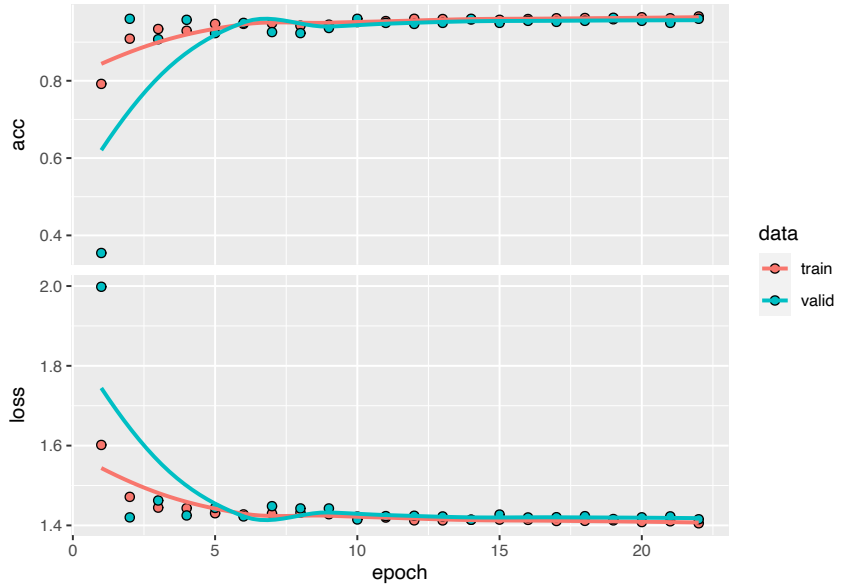


Figure 48: Training evolution of ResNet model.

Then, we classify a 16-year time series using the ResNet model. The behaviour of the ResNet model is similar to that of TempCNN, with more variability.

```
# Classify using DL model and plot the result
class <- point_mt_6bands %>%
   sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
   sits_classify(tempcnn_model) %>%
   plot(bands = c("NDVI", "EVI"))
```

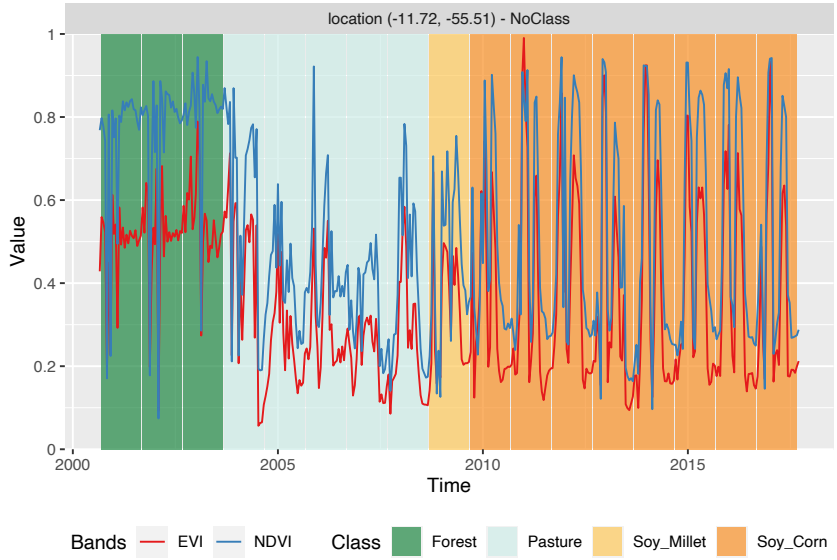


Figure 49: Classification of time series using ResNet.

Attention-based models

Attention-based models have become one of the most used deep learning architectures for problems that involve sequential data inputs, e.g., text recognition and automatic translation. The general idea is that in applications such as language translation not all inputs are alike. Consider the English sentence *"Look at all the lonely people"* and its Portuguese translation *"Olhe para todas as pessoas solitárias"*. A good translation system needs to relate the words "look" and "people" as the key parts of this sentence and to ensure such link is captured in the translation. A specific type of attention models, called *transformers*, enables the recognition of such complex relationships between input and output sequences [51].

The basic structure of transformers is the same as other neural network algorithms. They have an encoder that transforms the input textual values into numerical vectors, and a decoder that processes these vectors and provides suitable answers. The difference is on how the values are handled internally. In multilayer perceptrons (MLP), all inputs are treated equally at first; based on iterative matching of training and test data, the backpropagation technique feeds information back to the initial layers to identify the most relevant combination of inputs that produces the best output. In convolutional nets (CNN), input values that are close in time (1D) or space (2D) are combined to produce higher-level information that helps to distinguish the different components of the input data. For text recognition, the initial choice of deep learning studies was to use recurrent neural networks (RNN) that handle input sequences sequentially. However, neither MLPs, nor CNNs or RNNs have been able to capture the structure of complex inputs such as natural language. The success of transformer-based solutions accounts for substantial improvements in natural language processing.

The two main differences between transformer models and other algorithms are the use of positional encoding and self-attention. Positional encoding assigns an index to each input value which ensures that the relative locations of the inputs are maintained throughout the learning and processing phases. Self-attention works by comparing every word in the sentence to every other word in the sentence, including itself. In this way, it learns contextual information about the relation between the words. This conception has been validated in large language models such as BERT [52] and GPT-3 [53].

The application of attention-based models for satellite image time series analysis is proposed by Garnot et [31] and Russwurm and Körner [32]. The intuition behind the use of self-attention to image time series is to identify which observations are most relevant for classification. The first model proposed by Garnot and co-authors was a full transformer-based model [31]. Considering that image time series classification is a easier task that natural language processing, Garnot et al [54] also propose a simplified

version of the full transformer model. This simpler model uses a reduced way to compute the attention matrix, reducing time for training and classification without loss of quality of result.

In `sits`, the full transformer-based model proposed by Garnot and co-authors [31] is implemented using the `sits_tae()` function. The default parameters are those proposed by the authors. The default optimizer is the same is `optim_adamw`, available in package `torchopt`.

```
# train a machine learning model using TAE
tae_model <- sits_train(samples_matogrosso_mod13q1,
                  sits_tae(
                    epochs          = 150,
                    batch_size      = 64,
                    optimizer       = torchopt::optim_adamw,
                    validation_split = 0.2,
                    verbose         = FALSE) )
# show training evolution
plot(tae_model)
```

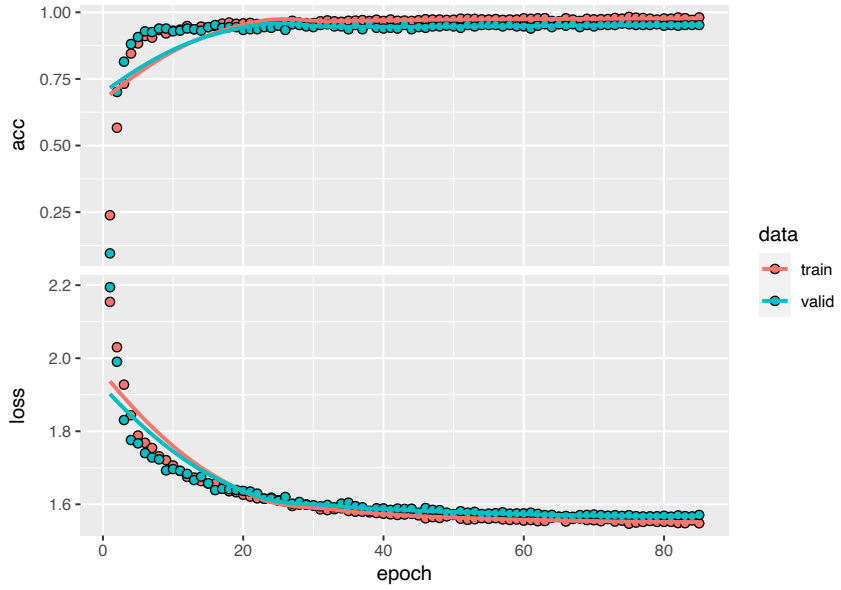


Figure 50: Training evolution of Temporal Self-Attention model.

Then, we classify a 16-year time series using the TAE model.

```
# Classify using DL model and plot the result
class <- point_mt_6bands %>%
   sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
   sits_classify(tae_model) %>%
   plot(bands = c("NDVI", "EVI"))
```

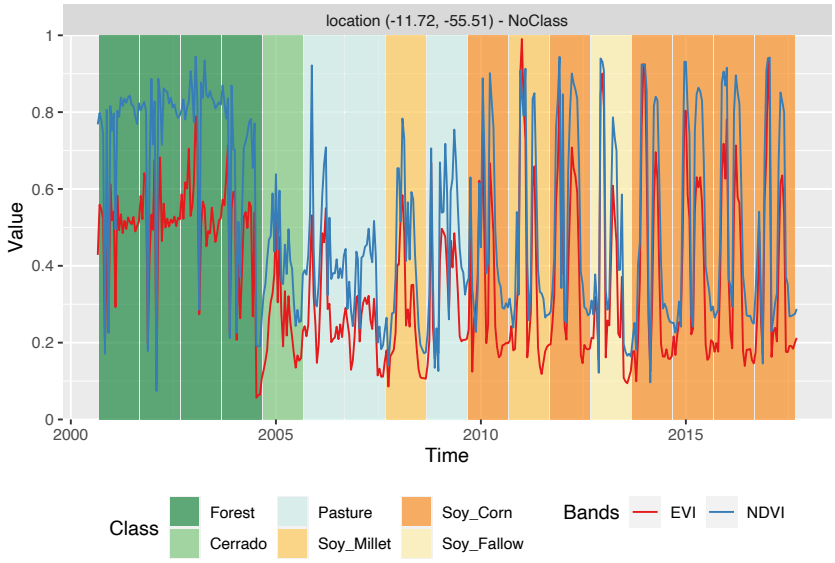


Figure 51: Classification of time series using TAE.

In `sits`, the simplified transformer-based model proposed by Garnot and co-authors [31] is implemented using the `sits_lighttae()` function. The default optimizer is the same is `optim_adamw`, available in package `torchopt`. The most important parameter to be set is the learning rate `lr`. Values ranging from 0.001 to 0.005 should produce reasonable results. See also the section below on model tuning.

```
# train a machine learning model using TAE
ltae_model <- sits_train(samples_matogrosso_mod13q1,
                 sits_lighttae(
                   epochs          = 150,
                   batch_size      = 64,
                   optimizer       = torchopt::optim_adamw,
                   opt_hparams = list(lr = 0.001),
                   validation_split = 0.2) )
# show training evolution
plot(ltae_model)
```

Then, we classify a 16-year time series using the LightTAE model.

```
# Classify using DL model and plot the result
class <- point_mt_6bands %>%
   sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
   sits_classify(ltae_model) %>%
   plot(bands = c("NDVI", "EVI"))
```

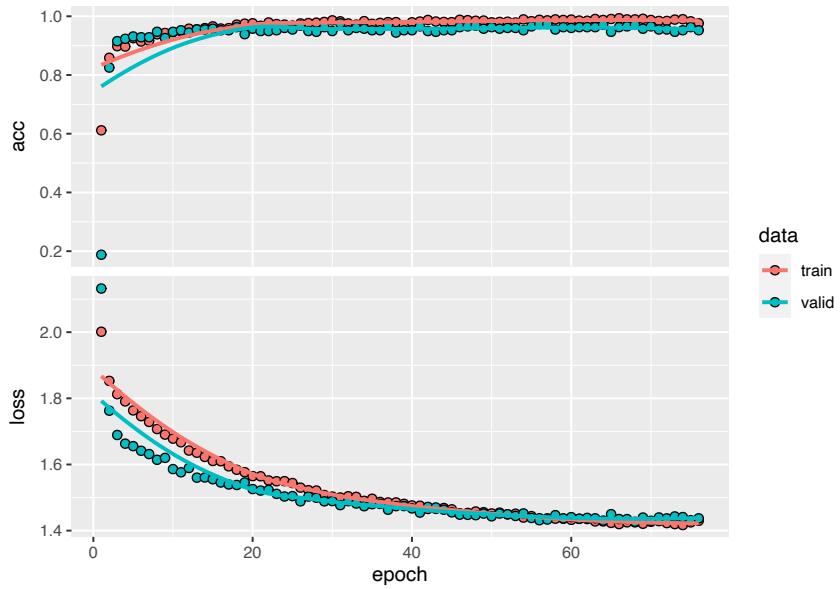


Figure 52: Training evolution of Lightweight Temporal Self-Attention model.

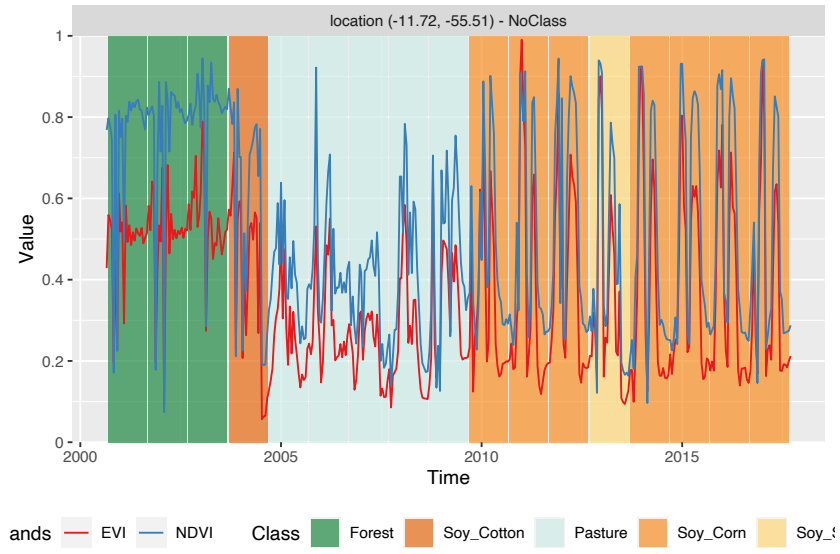


Figure 53: Classification of time series using LightTAE.

The behaviour of both `sits_tae()` and `sits_lighttae()` is similar to that of `sits_tempcnn()`. It points out the possible need for more classes and training data to better represent the transition period between 2004 and 2010. One possibility is that the training data associated to the Pasture class is only consistent with the time series between the years 2005 to 2008. However, the transition from Forest to Pasture in 2004 and from Pasture to Agriculture in 2009-2010 is subject to uncertainty, since the classifiers do not agree on the resulting classes. In general, the deep learning temporal-aware models are more sensitive to class variability than random forests and extreme gradient boosters.

Model tuning

Model tuning is an important step in machine learning classification, to enable a better fir of the method to the training data. Learning algorithms have a set of internal configuration called *hyperparameters*, which control the convergence of the algorithm and minimize the classification error. In practice, an unsuitable choice of hyperparameters can have a detrimental effect on the quality of the results.

Deep learning algorithms try to find the optimal point that represents the best value of the prediction function that, given an input X of data points, predicts the result Y. In our case, X is a multidimensional time series and Y is a vector of probabilities for the possible output classes. For complex situations, the best prediction function is time consuming to estimate. For this reason, deep learning methods rely on gradient descent methods to speed up predictions and converge faster than an exhaustive search [55]. All gradient descent methods use an optimization algorithm that is adjusted with hyperparameters such as the learning rate and regularization rate [56]. The learning rate controls the numerical step of the gradient descent function and the regularization rate controls model overfitting. Adjusting these values to an optimal setting requires the use of model tuning methods.

To reduce the learning curve, `sits` provides default values for all machine learning and deep learning methods described above. These defaults ensure a reasonable baseline performance. More experienced users may want to refine model hyperparameters, especially for more complex models such as `sits_lighttae()` or `sits_tempcnn()`. To that end, the package provides the `sits_tuning()` function.

The simplest approach to model tuning is would be to run a grid search; this involves defining a range for each hyperparameter and then testing all possible combinations. This approach leads to a combinational explosion and thus is not recommended. Instead, Bergstra and Bengio [57] propose to use randomly chosen trials. In their paper, the authors show that random trials are more efficient than grid search trials, allowing the selection of adequate hyperparameters at a fraction of the computational cost. The

`sits_tuning()` function follows Bergstra and Bengio [57] and uses a random search on the chosen hyperparameters.

Since gradient descent plays a key role in deep learning model fitting, developing optimizers is an important topic of research [58]. A large number of optimizers have been proposed in the literature, and recent results are reviewed by Schmidt et al [Schmidt2021]. For general deep learning applications, the *Adam* optimizer [59] provides a good baseline and reliable performance. For this reason, *Adam* is the default optimizer in the R `torch` package. However, experiments with image time series show that other optimizers may have better performance for the specific problem of land use classification. For this reason, the authors developed a separate R package, called `torchopt` that includes a number of recently proposed optimizers, including *Adamw* [60], *Madgrad* [61] and *Yogi* [62]. Based on our experiments, we have selected *Adamw* as the default optimizer for deep learning methods. Using the `sits_tuning()` function allows testing these and other optimizers available in `torch` and `torch_opt` packages.

The `sits_tuning()` function takes the following parameters:

1. `samples` - Training data set to be used by the model.
2. `samples_validation` (optional) - If available, this data set contains time series to be used for validation. If missing, the next parameter will be used.
3. `validation_split` - If `samples_validation` is not used, this parameter defines the proportion of time series in the training data set to be used for validation (default is 20%).
4. `ml_method()` - Deep learning method (either `sits_mlp()`, `sits_tempcnn()`, `sits_resnet()`, `sits_tae()` or `sits_lighttae()`)
5. `params` - defines the optimizer and its hyperparameters by calling the `sits_tuning_hparams()` function, as shown in the example below.
6. `trials` - number of trials to run the random search.
7. `multicores` - number of cores to be used for the procedure.
8. `progress` - show progress bar?

The `sits_tuning_hparams()` function inside `sits_tuning()` allows users to define optimizers and their hyperparameters including `lr` (learning rate), `eps` (controls numerical stability) and `weight_decay` (controls overfitting). The default values for `eps` and `weight_decay` in all `sits` deep learning functions are 1.0e-08 and 1.0e-06, respectively. The default `lr` for `sits_lighttae()` and `sits_tempcnn()` is 0.005, and for `sits_tae()` and `sits_resnet()` is 0.001. Users have different ways to randomize the hyperparameters, including: `choice()` (a list of options), `uniform` (a uniform distribution), `randint` (random integers from a uniform distribution), `normal(mean, sd)` (normal distribution), `beta(shape1, shape2)`(beta distribution). These options allow extensive combination of hyperparameters.

In the example, the `sits_tuning()` function finds good hyperparameters to train the `sits_lighttae()` method for the Mato Grosso data set. It tests 100 combinations of learning rate and weight decay for the *Adamw* optimizer. To randomize the learning rate, it uses a beta distribution with parameters 0.35 and 10, which allows for variation between about 0.2 and 1.0e-00; for the weight decay, the beta distribution with parameters 0.1 and 2 generates values roughly between 1 and 1.0e-24.

```
tuned <- sits_tuning(
    samples = samples_matogrosso_mod13q1,
    ml_method = sits_lighttae(),
    params = sits_tuning_hparams(
       optimizer = torchopt::optim_adamw,
       opt_hparams = list(
          lr = beta(0.35, 10),
          weight_decay = beta(0.1, 2)
       )
    ),
    trials = 100,
    multicores = 6,
    progress = FALSE
)
```

The result is a tibble with different values of accuracy, kappa, decision matrix, and hyperparameters. The 10 best results obtain accuracy values between 0.976 and 0.958, as shown below. The best result is obtained by a learning rate of 0.0011 and an weight decay of 2.14e-05,

```
# obtain accuracy, kappa, lr and weight decay for the 10
# best results hyperparameters are organized as a list
hparams_10 <- tuned[1:10, ]$opt_hparams
# extract learning rate and weight decay from the list
lr_10 <- purrr::map_dbl(hparams_10, function(h) h$lr)
wd_10 <- purrr::map_dbl(hparams_10, function(h) h$weight_decay)

# create a tibble to display the results
best_10 <- tibble::tibble(accuracy = tuned[1:10, ]$accuracy,
   kappa = tuned[1:10, ]$kappa, lr = lr_10, weight_decay = wd_10)
# print the best combination of hyperparameters
best_10
```

```
#> # A tibble: 10 x 4
#>   accuracy kappa  lr weight_decay
#>      <dbl> <dbl> <dbl>        <dbl>
#> 1    0.976 0.972 0.00117 2.14e- 5
#> 2    0.971 0.965 0.00125 2.07e- 4
```

```
#> 3   0.968 0.962 0.000281 2.70e- 2
#> 4   0.966 0.959 0.000418 1.54e- 2
#> 5   0.963 0.956 0.000432 1.76e- 6
#> 6   0.960 0.953 0.000263 2.35e- 4
#> 7   0.960 0.953 0.000254 3.13e- 3
#> 8   0.958 0.950 0.000973 1.35e- 2
#> 9   0.958 0.949 0.000694 8.95e-15
#> 10  0.958 0.950 0.000428 1.09e- 1
```

For large data sets, the tuning process is time consuming. Despite this cost, it is recommended for achieving the best performance. In general, tuning hyperparameters for models such as `sits_tempcnn()` and `sits_lighttae()` will result in a slight performance improvement over the default parameters on overall accuracy. The performance gain will be stronger in the less well represented classes, where significant gains in producer's and user's accuracies are possible. In cases where one wants to detect change in less frequent classes, tuning can make a difference in the results.

Considerations on model choice

The development of machine learning methods for classification of satellite image time series is an ongoing task. There is a lot of recent work using methods such as convolutional neural networks [30] and temporal self-attention [31]. Given the rapid evolution of the field with new methods still being developed, there are few references that offer a comparison between different machine learning methods. Most works on the literature [50] compare methods for generic time series classification. Their insights are not directly applicable for satellite image time series data, which have different properties than the time series using in applications such as economics and health.

In the specific case of satellite image time series, Russwurm et al. [32] present a comparative study between seven deep neural networks for classification of agricultural crops, using random forests (RF) as a baseline. The dataset is composed of Sentinel-2 images over Britanny, France. Their results indicate a slight difference between the best model (attention-based transformer model) over TempCNN, ResNet and RF. Attention-based models obtain accuracy ranging from 80-81%, TempCNN get 78-80%, and RF gets 78%. Based on this result and also on the authors' experience, we make the following recommendations:

1. Random forests provide a good baseline for image time series classification and should be included in users' assessments.

2. XGBoost is an worthy alternative to Random forests. In principle, XGBoost is more sensitive to data variations at the cost of possible overfitting.

3. TempCNN is a reliable model with reasonable training time, which is close to the state-of-the-art in deep learning classifiers for image time series.

4. Attention-based models (TAE and LightTAE) can achieve the best overall performance, in case of well designed and balanced training sets and with hyperparameter tuning.

5. The best means of improving classification performance is to provide an accurate and reliable training data set. Each class should have enough samples to account for spatial and temporal variability.

Classification of Images in Data Cubes using Satellite Image Time Series

---

This chapter shows the use of the SITS package for classification of satellite images that are associated to Earth observation data cubes.

Data cube classification

To classify a data cube, use the function `sits_classify()`. This function works in the same way for all types of data cubes, regardless of origin. When working with big EO data, the target environment for `sits` is a multicore virtual machine located close to the data repository.

To achieve efficiency, `sits` implements a fault tolerant multitasking procedure for big EO data classification. Users are not burdened with the need to learn how to do multiprocessing. Thus, their learning curve is shortened. Image classification in `sits` is done by a cluster of independent workers linked to a virtual machine. To avoid communication overhead, all large payloads are read and stored independently; direct interaction between the main process and the workers is kept at a minimum. The customised approach is depicted in the figure below.

1. Based on the size of the cube, the~number of cores, and~the available memory, divide the cube into chunks.
2. The cube is divided into chunks along its spatial dimensions. Each chunk contains all temporal intervals.
3. Assign chunks to the worker cores. Each core processes a block and produces an output image that is a subset of the result.

4. After all the subimages are produced, join them to obtain the result.
5. If a worker fails to process a block, provide failure recovery and ensure the worker completes the job.

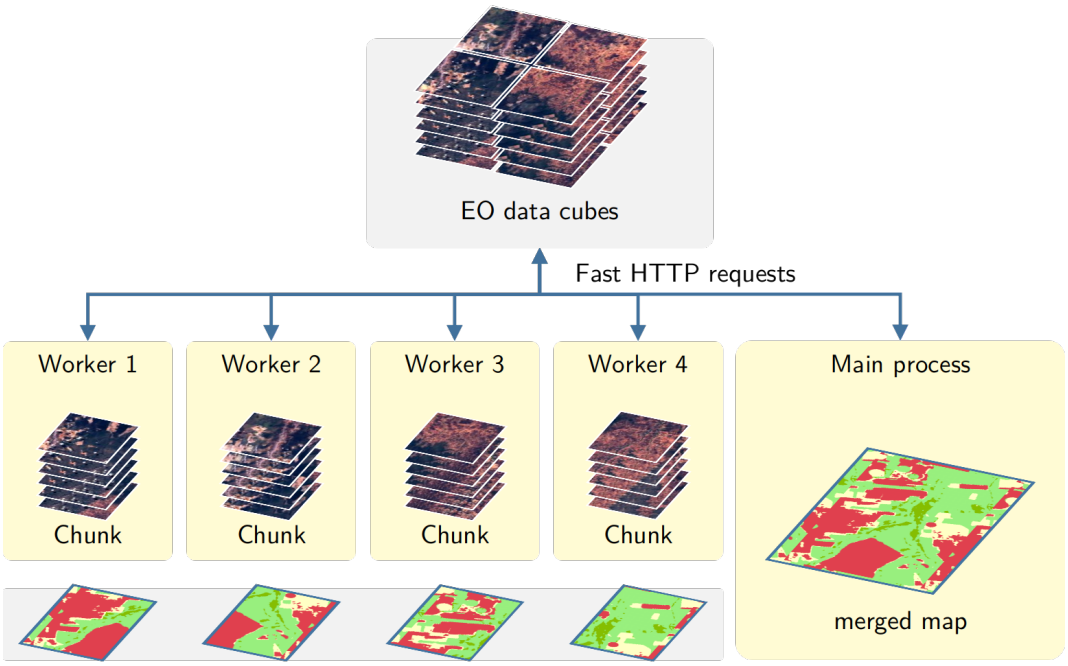


Figure 54: Parallel processing in sits (source: Simoes et al.(2021). Reproduction under fair use doctrine.

This approach has many advantages. It works in any virtual machine that supports R and has no dependencies on proprietary software. Processing is done in a concurrent and independent way, with no communication between workers. Failure of one worker does not cause failure of the big data processing. The~software is prepared to resume classification processing from the last processed chunk, preventing against failures such as memory exhaustion, power supply interruption, or network breakdown. From~an end-user point of view, all work is done smoothly and transparently.

The classification algorithm allows users to choose how many processes will run the task in parallel, and also the size of each data chunk to be consumed at each iteration. This strategy enables `sits` to work on average desktop computers without depleting all computational resources. The code bellow illustrates how to classify a large brick image that accompany the package.

To reduce processing time, it is necessary to adjust `sits_classify()` according to the capabilities of the host environment. There is a trade-off between computing time, memory use, and I/O operations. The best trade-off has to be determined by the user, considering issues such disk read speed, number of cores in the server, and CPU performance. The `memsize` parameter controls the size of the main memory (in GBytes) to

be used for classification. A practical approach is to set `memsize` to about 75% to 80% of the total memory available in the virtual machine. Users choose the number of cores to be used for parallel processing by setting the parameter `multicores`. We suggest that the `multicores` parameter is set to 1/4 of `memsize`.

Processing time estimates

Processing time depends on the data size and the model used. Some estimates derived from experiments made the authors show that:

1. Classification of one year of the entire Cerrado region of Brazil (2,5 million $km2$) using 18 tiles of CBERS-4 AWFI images (64 meter resolution), each tile consisting of 10,504 x 6,865 pixels with 24 time instances, using 4 spectral bands, 2 vegetation indexes and a cloud mask, resulting in 1,7 TB, took 16 hours using 100 GB of memory and 20 cores of a virtual machine. The classification was done with a random forest model with 2000 trees.

2. Classification of one year in one tile of LANDSAT-8 images (30 meter resolution), each tile consisting of 11,204 x 7,324 pixels with 24 time instances, using 7 spectral bands, 2 vegetation indexes and a cloud mask, resulting in 157 GB, took 90 minutes using 100 GB of memory and 20 cores of a virtual machine. The classification was done with a random forest model with 2000 trees.

3. The Brazilian Cerrado is covered by 51 Landsat-8 tiles available in the Brazil Data Cube (BDC). Each Landsat tile in the BDC covers a 3° $\times$ 2° grid in Albers equal area projection with an area of 73,920 km^2, and a size of 11,204 $\times$ 7324 pixels. The~one-year classification period ranges from September 2017 to August 2018, following the agricultural calendar. The~temporal interval is 16 days, resulting in 24 images per tile. We use seven spectral bands plus two vegetation indexes (NDVI and EVI) and the cloud cover information. The total input data size is about 8 TB. A data set of 48,850 samples was used to train a convolutional neural network model using the TempCNN method. All available attributes in the BDC Landsat-8 data cube (two vegetation indices and seven spectral bands) were used for training and classification. The classification was executed on an Ubuntu server with 24 cores and 128 GB memory. Each Landsat-8 tile was classified in an average of 30 min, and~the total classification took about 24 h.

Post-classification smoothing

Smoothing methods are an important complement to machine learning algorithms for image classification. Since these methods are mostly pixel-based, it is useful to complement them with post-processing smoothing to include spatial information in the result. For each pixel, machine learning and other statistical algorithms provide the probabilities of that pixel belonging to each of the classes. As a first step in obtaining a result, each pixel is assigned to the class whose probability is higher. After this step, smoothing methods use class probabilities to detect and correct outliers or misclassified pixels.

Image classification post-processing has been defined as "a refinement of the labelling in a classified image in order to enhance its classification accuracy" [63]. In remote sensing image analysis, these procedures are used to combine pixel-based classification methods with a spatial post-processing method to remove outliers and misclassified pixels. For pixel-based classifiers, post-processing methods enable the inclusion of spatial information in the final results.

Post-processing is a desirable step in any classification process. Most statistical classifiers use training samples derived from "pure" pixels, that have been selected by users as representative of the desired output classes. However, images contain many mixed pixels irrespective of the resolution. Also, there is a considerable degree of data variability in each class. These effects lead to outliers whose chance of misclassification is significant. To offset these problems, most post-processing methods use the "smoothness assumption" [64]: nearby pixels tend to have the same label. To put this assumption in practice, smoothing methods use the neighbourhood information to remove outliers and enhance consistency in the resulting product.

The following spatial smoothing methods are available in **sits**: bayesian smoothing, gaussian smoothing and bilinear smoothing. These methods are called using the `sits_smooth()` function, as shown in the examples below.

Bayesian smoothing

Bayesian inference can be thought of as way of coherently updating our uncertainty in the light of new evidence. It allows the inclusion of expert knowledge on the derivation of probabilities. In a Bayesian context, probability is taken as a subjective belief. The observation of the class probabilities of each pixel is taken as our initial belief on what the actual class of the pixel is. We then use Bayes' rule to consider how much the class probabilities of the neighboring pixels affect our original belief. In the case of continuous probability distributions, Bayesian inference is expressed by the rule:

$$\pi(\theta|x) \propto \pi(x|\theta)\pi(\theta)$$

Bayesian inference involves the estimation of an unknown parameter θ, which is the random variable that describe what we are trying to measure. In the case of smoothing of image classification, θ is the class probability for a given pixel. We model our initial belief about this value by a probability distribution, $\pi(\theta)$, called the *prior* distribution. It represents what we know about θ *before* observing the data. The distribution $\pi(x|\theta)$, called the *likelihood*, is estimated based on the observed data. It represents the added information provided by our observations. The *posterior* distribution $\pi(\theta|x)$ is our improved belief of θ *after* seeing the data. Bayes's rule states that the *posterior* probability is proportional to the product of the *likelihood* and the *prior* probability.

Derivation of bayesian parameters for spatiotemporal smoothing

In our post-classification smoothing model, we consider the output of a machine learning algorithm that provides the probabilities of each pixel in the image to belong to target classes. More formally, consider a set of K classes that are candidates for labelling each pixel. Let $p_{i,t,k}$ be the probability of pixel i belonging to class k, $k = 1, \dots, K$ at a time t, $t = 1, \dots, T$. We have

$$\sum_{k=1}^{K} p_{i,t,k} = 1, p_{i,t,k} > 0$$

We label a pixel p_i as being of class k if

$$p_{i,t,k} > p_{i,t,m}, \forall m = 1, \dots, K, m \neq k$$

For each pixel i, we take the odds of the classification for class k, expressed as

$$O_{i,t,k} = p_{i,t,k}/(1 - p_{i,t,k})$$

where $p_{i,t,k}$ is the probability of class k at time t. We have more confidence in pixels with higher odds since their class assignment is stronger. There are situations, such as border pixels or mixed ones, where the odds of different classes are similar in magnitude. We take them as cases of low confidence in the classification result. To assess and correct these cases, Bayesian smoothing methods borrow strength from the neighbors and reduces the variance of the estimated class for each pixel.

We further make the transformation

$$x_{i,t,k} = \log[O_{i,t,k}]$$

which measures the *logit* (log of the odds) associated to classifying the pixel i as being of class k at time t. The support of $x_{i,t,k}$ is $\mathbb{R}$. We can express the pixel data as a K-dimensional multivariate logit vector

$$\mathbf{x}_{i,t} = (x_{i,t,k_0}, x_{i,t,k_1}, \dots, x_{i,t,k_K})$$

For each pixel, the random variable that describes the class probability k at time t is denoted by $\theta_{i,t,k}$. This formulation allows uses to use the class covariance matrix in our formulations. We can express Bayes' rule for all combinations of pixel and classes for a time interval as

$$\pi(\theta_{i,t}|\mathbf{x}_{i,t}) \propto \pi(\mathbf{x}_{i,t}|\theta_{i,t})\pi(\theta_{i,t}).$$

We assume the conditional distribution $\mathbf{x}_{i,t}|\theta_{i,t}$ follows a multivariate normal distribution

$$[\mathbf{x}_{i,t}|\theta_{i,t}] \sim \mathcal{N}_K(\theta_{i,t}, \Sigma_{i,t}),$$

where $\theta_{i,t}$ is the mean parameter vector for the pixel i at time t, and $\Sigma_{i,t}$ is a known $k \times k$ covariance matrix that we will use as a parameter to control the level of smoothness effect. We will discuss later on how to estimate $\Sigma_{i,t}$. To model our uncertainty about the parameter $\theta_{i,t}$, we will assume it also follows a multivariate normal distribution with hyper-parameters $\mathbf{m}_{i,t}$ for the mean vector, and $\mathbf{S}_{i,t}$ for the covariance matrix.

$$[\theta_{i,t}] \sim \mathcal{N}_K(\mathbf{m}_{i,t}, \mathbf{S}_{i,t}).$$

The above equation defines our prior distribution. The hyper-parameters $\mathbf{m}_{i,t}$ and $\mathbf{S}_{i,t}$ are obtained by considering the neighboring pixels of pixel i. The neighborhood can be defined as any graph scheme (e.g. a given Chebyshev distance on the time-space lattice) and can include the referencing pixel i as a neighbor. Also, it can make no reference to time steps other than t defining a space-only neighborhood. More formally, let

$$\mathbf{V}_{i,t} = \{\mathbf{x}_{i_j,t_j}\}_{j=1}^{N},$$

denote the N logit vectors of a spatiotemporal neighborhood N of pixel i at time t. Then the prior mean is calculated by

$$\mathbf{m}_{i,t} = \mathrm{E}[\mathbf{V}_{i,t}],$$

and the prior covariance matrix by

$$\mathbf{S}_{i,t} = \mathrm{E}\left[\left(\mathbf{V}_{i,t} - \mathbf{m}_{i,t}\right)\left(\mathbf{V}_{i,t} - \mathbf{m}_{i,t}\right)^{\top}\right].$$

Since the likelihood and prior are multivariate normal distributions, the posterior will also be a multivariate normal distribution, whose updated parameters can be derived by applying the density functions associated to the above equations. The posterior distribution is given by

$$[\theta_{i,t}|\mathbf{x}_{i,t}] \sim \mathcal{N}_K\left((\mathbf{S}_{i,t}^{-1} + \Sigma^{-1})^{-1}(\mathbf{S}_{i,t}^{-1}\mathbf{m}_{i,t} + \Sigma^{-1}\mathbf{x}_{i,t}), (\mathbf{S}_{i,t}^{-1} + \Sigma^{-1})^{-1}\right).$$

The $\theta_{i,t}$ parameter model is our initial belief about a pixel vector using the neighborhood information in the prior distribution. It represents what we know about the probable value of $\mathbf{x}_{i,t}$ (and hence, about the class probabilities as the logit function is a monotonically increasing function) *before* observing it. The *likelihood* function $P[\mathbf{x}_{i,t}|\theta_{i,t}]$ represents the added information provided by our observation of $\mathbf{x}_{i,t}$. The *posterior* probability density function $P[\theta_{i,t}|\mathbf{x}_{i,t}]$ is our improved belief of the pixel vector *after* seeing $\mathbf{x}_{i,t}$.

At this point, we are able to infer a point estimator $\hat{\theta}_{i,t}$ for the $\theta_{i,t}|\mathbf{x}_{i,t}$ parameter. For the multivariate normal distribution, the posterior mean minimises not only the quadratic loss but the absolute and zero-one loss functions. It can be taken from the updated mean parameter of the posterior distribution which, after some algebra, can be expressed as

$$\hat{\theta}_{i,t} = \mathrm{E}[\theta_{i,t}|\mathbf{x}_{i,t}] = \Sigma_{i,t}\left(\Sigma_{i,t} + \mathbf{S}_{i,t}\right)^{-1}\mathbf{m}_{i,t} + \mathbf{S}_{i,t}\left(\Sigma_{i,t} + \mathbf{S}_{i,t}\right)^{-1}\mathbf{x}_{i,t}.$$

The estimator value for the logit vector $\hat{\theta}_{i,t}$ is a weighted combination of the original logit vector $\mathbf{x}_{i,t}$ and the neighborhood mean vector $\mathbf{m}_{i,t}$. The weights are given by the covariance matrix $\mathbf{S}_{i,t}$ of the prior distribution and the covariance matrix of the conditional distribution. The matrix $\mathbf{S}_{i,t}$ is calculated considering the spatiotemporal neighbors and the matrix $\Sigma_{i,t}$ corresponds to the smoothing factor provided as prior belief by the user.

When the values of local class covariance $\mathbf{S}_{i,t}$ are relative to the conditional covariance $\Sigma_{i,t}$, our confidence on the influence of the neighbors is low, and the smoothing algorithm gives more weight to the original pixel value $x_{i,k}$. When the local class covariance $\mathbf{S}_{i,t}$ decreases relative to the smoothness factor $\Sigma_{i,t}$, then our confidence on the influence of the neighborhood increases. The smoothing procedure will be most relevant in situations where the original classification odds ratio is low, showing a low level of separability between classes. In these cases, the updated values of the classes will be influenced by the local class variances.

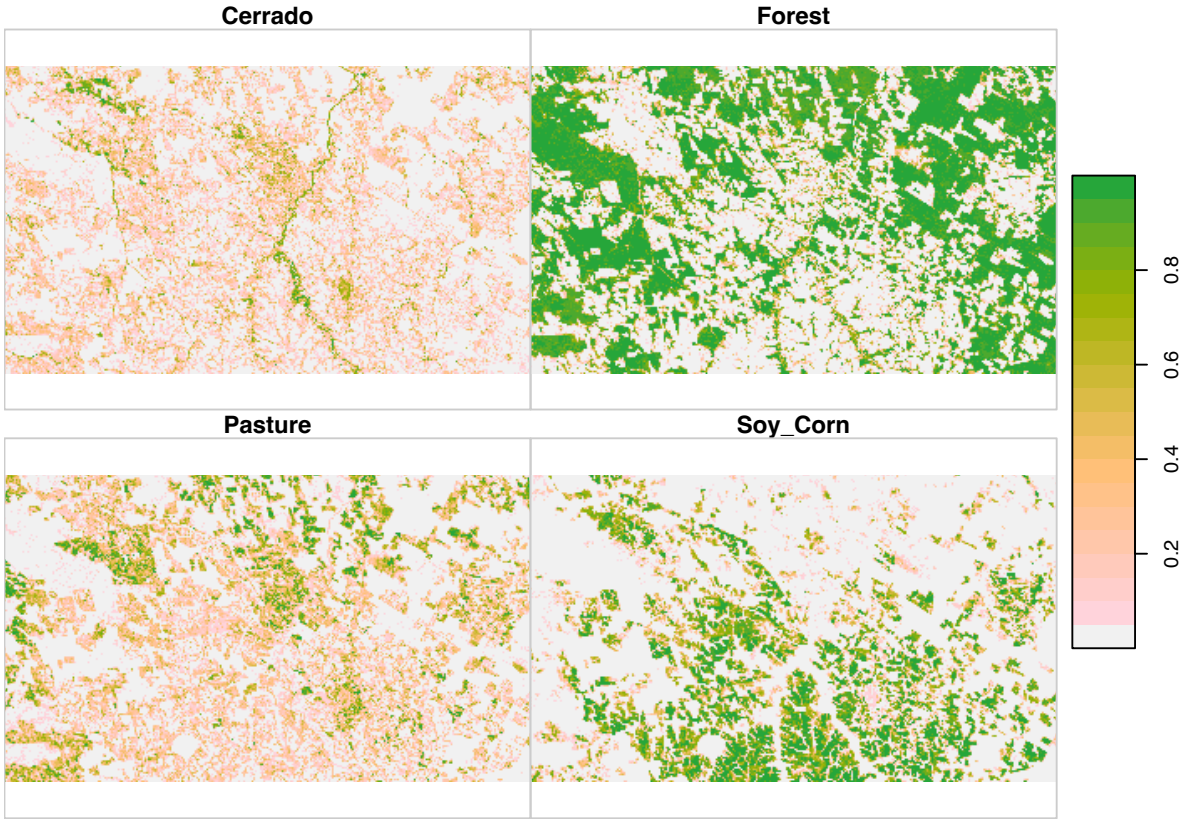
In practice, $\Sigma_{i,t}$ is a user-controlled covariance matrix parameter that will be set by users based on their knowledge of the region to be classified. In the simplest case, users can associate the conditional covariance $\Sigma_{i,t}$ to a diagonal matrix, using only one hyperparameter σ_k^2 to set the level of smoothness. Higher values of σ_k^2 will cause the assignment of the local mean to the pixel updated probability. In our case, after some classification tests, we decided to $\sigma_k^2 = 20$ by default for all k.

Use of Bayesian smoothing in SITS

Doing post-processing using Bayesian smoothing in SITS is straightforward. The result of the `sits_classify` function applied to a data cube is set of probability images, one

per class. The next step is to apply the `sits_smooth` function. By default, this function selects the most likely class for each pixel considering only the probabilities of each class for each pixel. To allow for Bayesian smoothing, it suffices to include the `type = bayesian` parameter (which is also the default). If desired, the `smoothness` parameter (associated to the hyperparameter σ_k^2 described above) can control the degree of smoothness. If so desired, the `smoothness` parameter can also be expressed as a matrix

```
# Retrieve the data for the Mato Grosso state
data("samples_modis_4bands")

# select the bands 'ndvi', 'evi'
samples_2bands <- sits_select(samples_modis_4bands, bands = c("NDVI",
   "EVI"))

# select a rfor model
rfor_model <- sits_train(samples_2bands, ml_method = sits_rfor())

data_dir <- system.file("extdata/sinop", package = "sitsdata")

# create a raster metadata file based on the information
# about the files
raster_cube <- sits_cube(source = "BDC", collection = "MOD13Q1-6",
   data_dir = data_dir, parse_info = c("X1", "X2", "tile", "band",
      "date"))
# classify the raster image and generate a probability file
raster_probs <- sits_classify(raster_cube, ml_model = rfor_model,
   memsize = 12, multicores = 3, output_dir = "./tempdir/chp8")

plot(raster_probs)
```



The plots show the class probabilities, which can then be smoothed by a bayesian smoother.

```
# smooth the result with a bayesian filter
raster_probs_bayes <- sits_smooth(raster_probs, type = "bayes",
   memsize = 12, multicores = 3, output_dir = "./tempdir/chp8")
# plot the result
plot(raster_probs_bayes)
```

The bayesian smoothing has removed some of local variability associated to misclassified pixels which are different from their neighbors. The impact of smoothing is best appreciated comparing the labelled map produced without smoothing to the one that follows the procedure, as shown below.

The resulting labelled map shows a number of likely misclassified pixels which can be removed using the bayesian smoother.

Comparing the two plots, it is apparent that the smoothing procedure has reduced a lot of the noise in the original classification and produced a more homogeneous result.

Bilateral smoothing

One of the problems with post-classification smoothing is that we would like to remove noisy pixels (e.g., a pixel with high probability of being labeled "Forest" in the midst

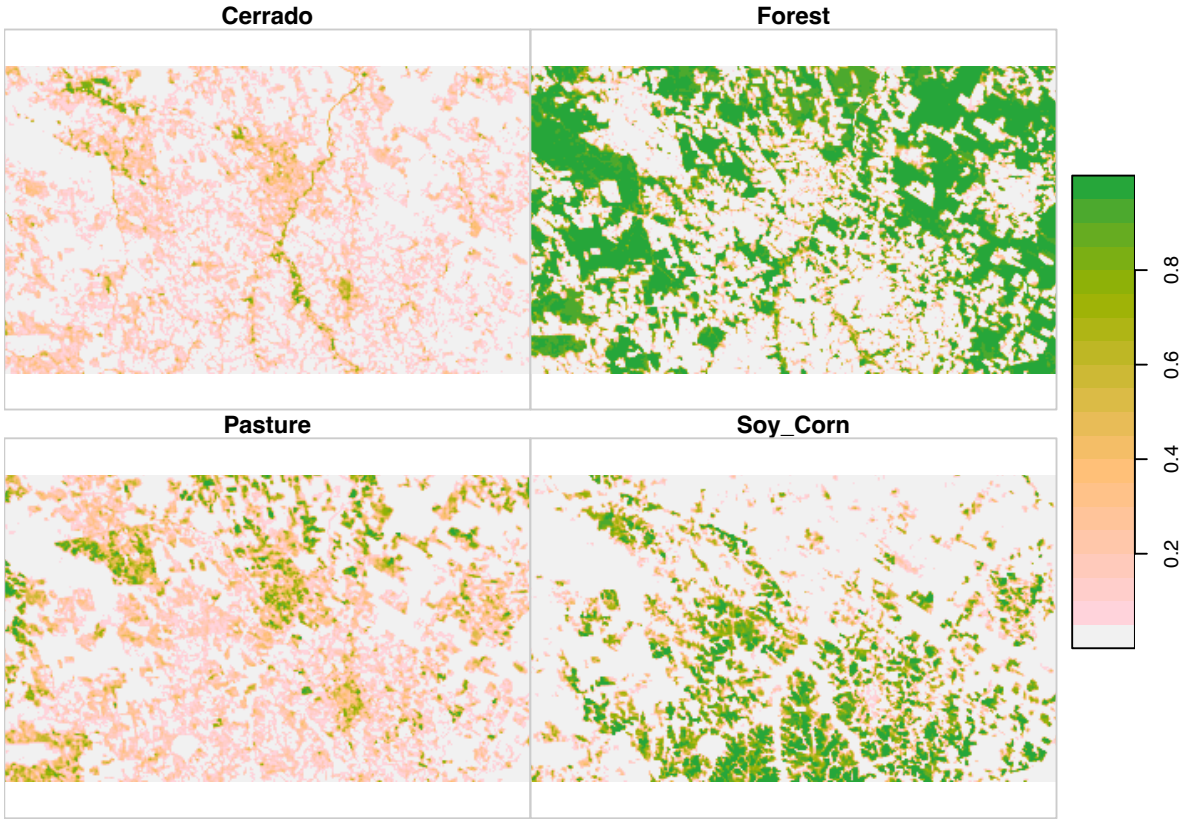


Figure 55: Probability values smoothed by bayesian method

of pixels likely to be labeled “Cerrado”), but would like to preserve the edges between areas. Because of its design, bilateral filter has proven to be a useful method for post-classification processing since it preserves edges while removing noisy pixels [64].

Bilateral smoothing combines proximity (combining pixels which are close) and similarity (comparing the values of the pixels) [65]. If most of the pixels in a neighborhood have similar values, it is easy to identify outliers and noisy pixels. In contrast, there is a strong difference between the values of pixels in a if neighborhood, it is possible that the pixel is located in a class boundary. Bilateral filtering combines domain filtering with range filtering. In domain filtering, the weights used to combine pixels decrease with distance. In range filtering, the weights are computed considering value similarity.

The combination of domain and range filtering is mathematically expressed as:

$$S(x_i) = \frac{1}{W_i} \sum_{x_k \in \theta} I(x_k) \mathcal{N}_\tau(\|I(x_k) - I(x_i)\|) \mathcal{N}_\sigma(\|x_k - x_i\|),$$

where

- $S(x_i)$ is the smoothed value of pixel i;
- I is the original probability image to be filtered;

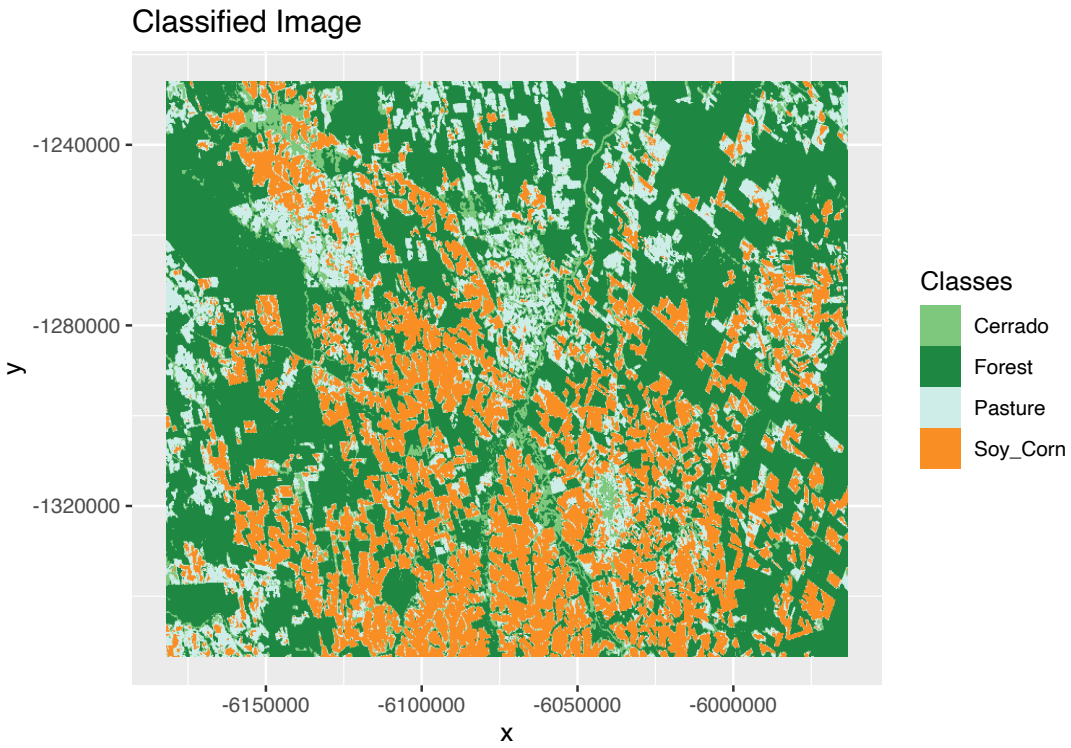


Figure 56: Classified image without smoothing

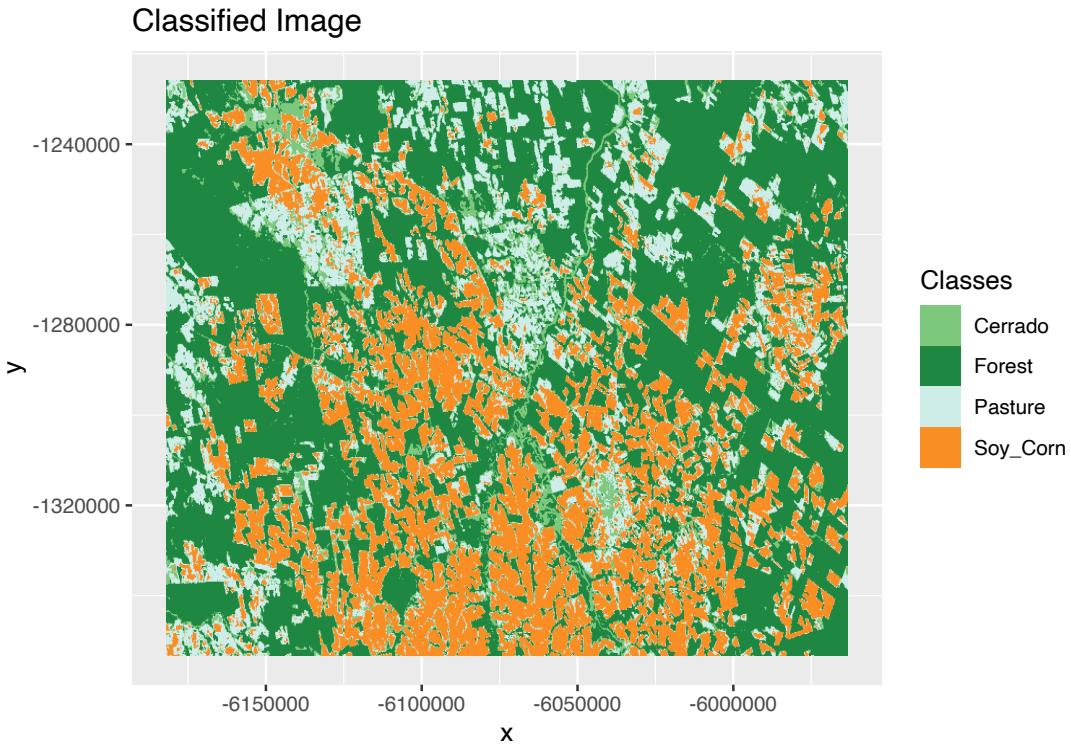


Figure 57: Classified image with Bayesian smoothing

- $I(x_i)$ is the value of pixel i;
- θ is the neighborhood centered in x_i;
- x_k is a pixel k which belongs to neighborhood θ;
- $I(x_k)$ is the value of a pixel k in the neighborhood of pixel i;
- $\|I(x_k) - I(x_i)\|$ is the absolute difference between the values of the pixel k and pixel i;
- $\|x_k - x_i\|$ is the distance between pixel k and pixel i;
- $\mathcal{N}_\tau$ is the Gaussian range kernel for smoothing differences in intensities;
- $\mathcal{N}_\sigma$is the Gaussian spatial kernel for smoothing differences based on proximity.
- τ is the variance of the Gaussian range kernel;
- σ is the variance of the Gaussian spatial kernel.

The normalization term to be applied to compute the smoothed values of pixel i is defined as

$$W_i = \sum_{x_k \in \theta} \mathcal{N}_\tau(\|I(x_k) - I(x_i)\|)\mathcal{N}_\sigma(\|x_k - x_i\|)$$

For every pixel, the method takes a considers two factors: the distance between the pixel and its neighbors, and the difference in value between them. Each of the values contributes according to a Gaussian kernel. These factors are calculated independently. Big difference between pixel values reduce the influence of the neighbor in the smoothed pixel. Big distance between pixels also reduce the impact of neighbors. The achieve a satisfactory result, we need to balance the σ and τ. As a general rule, the values of τ should range from 0.05 to 0.50, while the values of σ should vary between 4 and 16[66]. The default values adopted in *sits* are `tau = 0.1` and `sigma = 8`. As the best values of τ and σ depend on the variance of the noisy pixels, users are encouraged to experiment and find parameter values that best fit their requirements.

The following example shows the behavior of the bilateral smoother.

```
# smooth the result with a bilateral filter
raster_probs_bil <- sits_smooth(raster_probs, type = "bilateral",
   sigma = 8, tau = 0.1, output_dir = "./tempdir/chp8")
# plot the result
plot(raster_probs_bil)
```

The impact on the classified image can be seen in the following example.

Bayesian smoothing tends to produce more homogeneous labeled images than bilateral smoothing. However, some spatial details and some edges are better preserved by the bilateral method. Choosing between the methods depends on user needs and requirements. In any case, as stated by [64], smoothing improves the quality of classified images and thus should be applied in most situations.

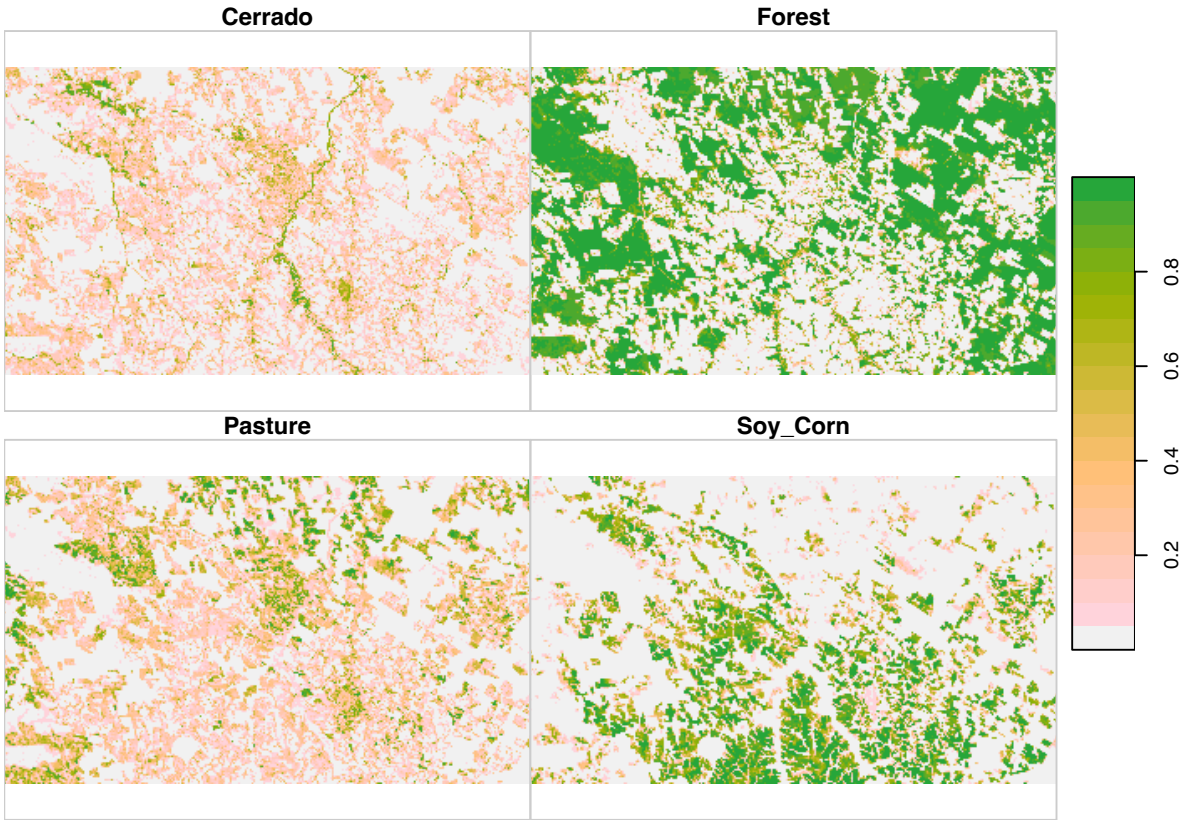


Figure 58: Probability values for classified image smoothed by bilateral filter

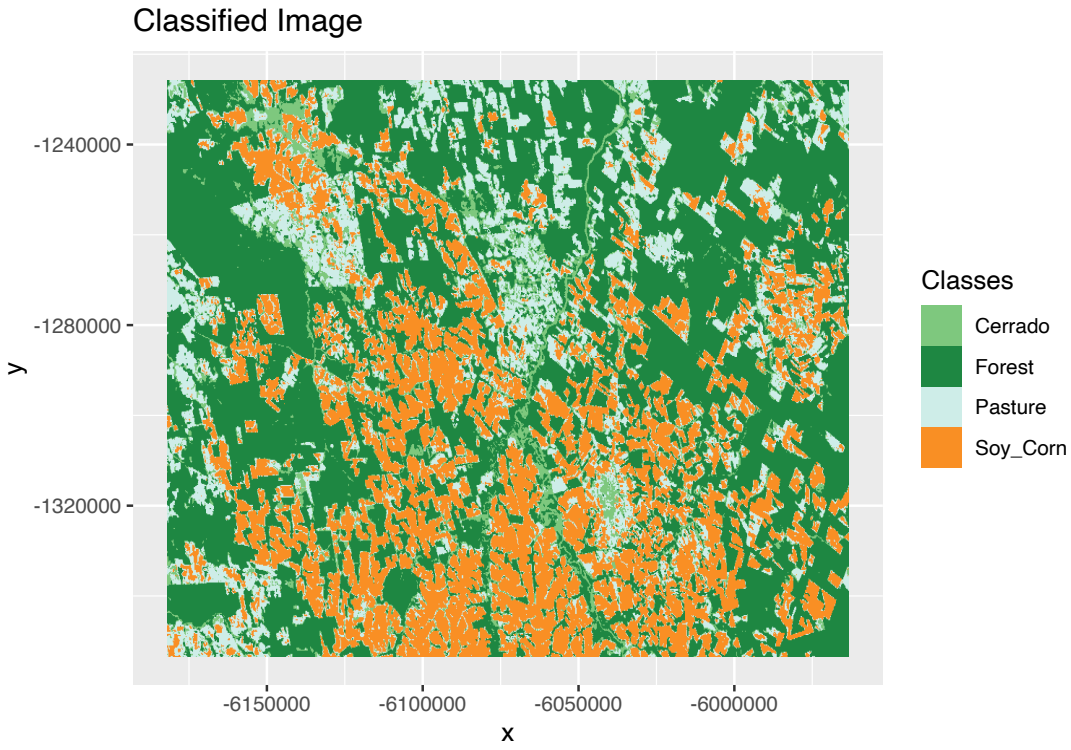


Figure 59: Classified image with bilateral smoothing

Validation and accuracy measurements in SITS

This chapter presents the validation and accuracy measures available in the SITS package.

Validation techniques

Validation is a process undertaken on models to estimate some error associated with them, and hence has been used widely in different scientific disciplines. When we talk about validation, we are interested in estimating the prediction error associated to some model. For this purpose, we concentrate on the *cross-validation* approach, probably the most used validation technique [39].

Notice that validation techniques are not a replacement of accuracy measures, which are described below. Validation methods are based on the training samples. In general, these samples will be biased due to many factors. When working in large areas, it is hard to obtain random stratified samples which would cover the different variations in land cover associated to the ecosystems of the study area. In general, all machine learning methods are prone to underspecification [67]. Therefore, cross-validation should not be used as an accuracy measures, unless the samples have been carefully collected to represent the diversity of possible ocurrences of classes in the study area [68].

Cross-validation uses part of the available samples to fit the classification model, and a different part to test it. The so-called *k-fold* validation, we split the data into k partitions with approximately the same size and proceed by fitting the model and testing it k times. At each step, we take one distinct partition for test and the remaining $k-1$ for training the model, and calculate its prediction error for classifying the test partition. A simple average gives us an estimation of the expected prediction error.

A natural question that arises is: *how good is this estimation?* According to [39], there is a bias-variance trade-off in choice of k. If k is set to the number of samples, we obtain the so-called *leave-one-out* validation, the estimator gives a low bias for the true expected error, but produces a high variance expectation. This can be computational expensive as it requires the same number of fitting process as the number of samples. On the other hand, if we choose $k = 2$, we get a high biased expected prediction error estimation that overestimates the true prediction error, but has a low variance. The recommended choices of k are 5 or 10.

`sits_kfold_validate()` gives support the k-fold validation in `sits`. The following code gives an example on how to proceed a k-fold cross-validation in the package. It perform a five-fold validation using SVM classification model as a default classifier. We can see in the output text the corresponding confusion matrix and the accuracy statistics (overall and by class).

```
# perform a five fold validation for the 'cerrado_2classes'
# data set Random Forest machine learning method using
# default parameters
val_rfor <- sits_kfold_validate(cerrado_2classes, folds = 5,
   ml_method = sits_rfor())
# print the validation statistics
val_rfor
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction Cerrado Pasture
#>   Cerrado  394    14
#>   Pasture    6    332
#>
#>          Accuracy : 0.9732
#>            95% CI : (0.9589, 0.9835)
#>
#>             Kappa : 0.946
#>
#> Prod Acc Cerrado : 0.9850
#> Prod Acc Pasture : 0.9595
#> User Acc Cerrado : 0.9657
#> User Acc Pasture : 0.9822
#>
```

Comparing different machine learning methods using k-fold validation

One useful function in SITS is the capacity to compare different validation methods and store them in an XLS file for further analysis. The following example shows how to do this, using the Mato Grosso data set. We take five models: random forests(`sits_rfor()`), support vector machines (`sits_svm()`), extreme gradient boosting (`sits_xgboost()`), multi-layer perceptron (`sits_mlp()`) and temporal convolutional neural network (`sits_tempcnn()`). For simplicity, we use the default parameters provided by sits. After computing the confusion matrix and the statistics for each model, we store the result in a list. When the calculation is finished, the function `sits_to_xlsx` writes all of the results in an Excel-compatible spreadsheet.

```
# Retrieve the set of samples for the Mato Grosso region
# (provided by EMBRAPA)
data("samples_matogrosso_mod13q1")
# create a list to store the results
results <- list()

# SVM model
conf_svm <- sits_kfold_validate(samples_matogrosso_mod13q1, ml_method =
    sits_svm())
# Give a name to the SVM model
conf_svm$name <- "svm"
# store the result
results[[length(results) + 1]] <- conf_svm

# Random Forest
conf_rfor <- sits_kfold_validate(samples_matogrosso_mod13q1,
   folds = 5, multicores = 2, ml_method = sits_rfor())
# Give a name to the results
conf_rfor$name <- "rfor"
# store the results in a list
results[[length(results) + 1]] <- conf_rfor

## Extreme Gradient Boosting
conf_xgb <- sits_kfold_validate(samples_matogrosso_mod13q1, ml_method =
    sits_xgboost())

# Give a name to the SVM model
conf_xgb$name <- "xgboost"
# store the results in a list
results[[length(results) + 1]] <- conf_xgb
```

```
# Multi-layer perceptron
conf_mlp <- sits_kfold_validate(samples_matogrosso_mod13q1, ml_method =
    sits_mlp())

# Give a name to the SVM model
conf_mlp$name <- "MLP"
# store the results in a list
results[[length(results) + 1]] <- conf_mlp

# Temporal CNN
conf_tcnn <- sits_kfold_validate(samples_matogrosso_mod13q1,
   ml_method = sits_tempcnn())

# Give a name to the SVM model
conf_tcnn$name <- "TempCNN"
# store the results in a list
results[[length(results) + 1]] <- conf_tcnn

xlsx_file <- paste0(getwd(), "/model_comparison.xlsx")
# Save to an XLS file
sits_to_xlsx(results, file = xlsx_file)
```

The resulting Excel file can be opened with R or using spreadsheet programs. The figure below shows a printout of what is read by Excel. As shown below, each sheet corresponds to the output of one model. For simplicity, we show only the result of TempCNN, that has an overall accuracy of 97% and is the best-performing model.

Accuracy assessment

Time series

Users can perform accuracy assessment in *sits* both in time series datasets or in classified images using the `sits_accuracy` function. In the case of time series, the input is a sits tibble which has been classified by a sits model. The input tibble needs to contain valid labels in its "label" column. These labels are compared to the results of the prediction to the reference values. This function calculates the confusion matrix and then the resulting statistics using the R package "caret".

```
# read a tibble with 400 time series of Cerrado and 346 of
# Pasture
data(cerrado_2classes)
# create a model for classification of time series
```

	A	B	C	D	E	F	G	H	I	J
1		Pasture	Soy_Corn	Soy_Millet	Soy_Cottor	Fallow_Cot	Soy_Sunflow	Cerrado	Forest	Soy_Fallow
2	Pasture	340	3	1	2	1	0	2	0	0
3	Soy_Corn	0	340	6	8	0	3	0	0	0
4	Soy_Millet	0	13	171	0	0	0	0	0	1
5	Soy_Cotton	0	5	0	341	1	0	0	0	0
6	Fallow_Cotton	0	0	0	1	27	0	0	0	0
7	Soy_Sunflower	0	3	0	0	0	23	0	0	0
8	Cerrado	4	0	1	0	0	0	377	0	0
9	Forest	0	0	0	0	0	0	0	131	0
10	Soy_Fallow	0	0	1	0	0	0	0	0	86
11										
12		V1								
13	Accuracy	0.97								
14	Kappa	0.96								
15										
16										
17		Pasture	Soy_Corn	Soy_Millet	Soy_Cottor	Fallow_Cot	Soy_Sunflow	Cerrado	Forest	Soy_Fallow
18	Sensitivity (PA)	0.99	0.93	0.95	0.97	0.93	0.88	0.99	1.00	0.99
19	Specificity	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
20	PosPredValue (UA)	0.97	0.95	0.92	0.98	0.96	0.88	0.99	1.00	0.99
21	NegPredValue	1.00	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00
22										

svm | rfor | xgboost | MLP | TempCNN | +

Figure 60: Result of 5-fold cross validation of Mato Grosso dataset using TempCNN

```
svm_model <- sits_train(cerrado_2classes, sits_svm())
# classify the time series
predicted <- sits_classify(cerrado_2classes, svm_model)
# calculate the classification accuracy
acc_ts <- sits_accuracy(predicted)
# print the accuracy statistics summary
sits_accuracy_summary(acc_ts)
```

```
#> Overall Statistics
#> Accuracy : 0.9906
#>   95% CI : (0.9808, 0.9962)
#>    Kappa : 0.9811
```

The detailed accuracy measures can be obtained by printing the accuracy object.

```
# print the accuracy statistics
acc_ts
```

```
#> Confusion Matrix and Statistics
#>
#>          Reference
#> Prediction Cerrado Pasture
#>   Cerrado  397     4
#>   Pasture    3    342
#>
#>          Accuracy : 0.9906
#>            95% CI : (0.9808, 0.9962)
```

```
#>
#>           Kappa : 0.9811
#>
#> Prod Acc Cerrado : 0.9925
#> Prod Acc Pasture : 0.9884
#> User Acc Cerrado : 0.9900
#> User Acc Pasture : 0.9913
#>
```

Classified images

To measure the accuracy of classified images, the `sits_accuracy` function uses an area-weighted technique, following the best practices proposed by [69]. The need for area-weighted estimates arises from the fact the land use and land cover classes are not evenly distributed in space. In some applications (e.g., deforestation) where the interest lies in assessing how much of the image has changed, the area mapped as deforested is likely to be a small fraction of the total area. If users disregard the relative importance of small areas where change is taking place, the overall accuracy estimate will be inflated and unrealistic. For this reason, [69] argue that "mapped areas should be adjusted to eliminate bias attributable to map classification error and these error-adjusted area estimates should be accompanied by confidence intervals to quantify the sampling variability of the estimated area".

With this motivation, when measuring accuracy of classified images, the function `sits_accuracy` follows [69] and [70]. The following explanation is extracted from the paper of [69], and users should refer to this paper for further explanation.

Given a classified image and a validation file, the first step is to calculate the confusion matrix in the traditional way, i.e., by identifying the commission and omission errors. Then we calculate the unbiased estimator of the proportion of area in cell i, j of the error matrix

$$\hat{p_{i,j}} = W_i \frac{n_{i,j}}{n_i}$$

where the total area of the map is A_{tot}, the mapping area of class i is $A_{m,i}$ and the proportion of area mapped as class i is $W_i = A_{m,i}/A_{tot}$. Adjusting for area size allows producing an unbiased estimation of the total area of class j, defined as a stratified estimator

$$\hat{A}_j = A_{tot} \sum_{i=1}^{K} W_i \frac{n_{i,j}}{n_i}$$

This unbiased area estimator includes the effect of false negatives (omission error) while not considering the effect of false positives (commission error). The area estimates also allow producing an unbiased estimate of the user's and producer's accuracy for each class. Following [69], we can also estimate the 95% confidence interval for $\hat{A}_j$.

To use the sits_accuracy function to produce the adjusted area estimates, users have to provide the classified image together with a csv file containing a set of well selected labeled points. The csv file should have the same format as the one used to obtain samples, as discussed earlier. The labelled points should be based on a random stratified sample. All areas associated to each class should contribute to the test data used for accuracy asseesment.

In what follows, we show a simple example of using the accuracy function to estimate the quality of the classification

```
# select a sample with 2 bands (NDVI and EVI)
samples_modis_ndvi <- sits_select(samples_modis_4bands, bands = "NDVI")
# build an extreme gradient boosting model
xgb_model <- sits_train(samples_modis_ndvi, ml_method = sits_xgboost())
# create a data cube based on files
data_dir <- system.file("extdata/raster/mod13q1", package = "sits")
cube <- sits_cube(source = "BDC", collection = "MOD13Q1-6", name = "Sinop
    ",
   data_dir = data_dir, parse_info = c("X1", "X2", "tile", "band",
     "date"))
# classify the data cube with xgb model
probs_cube <- sits_classify(cube, xgb_model)
# label the classification
label_cube <- sits_label_classification(probs_cube, output_dir = "./
    tempdir/chp9")
# get ground truth points
ground_truth <- system.file("extdata/samples/samples_sinop_crop.csv",
   package = "sits")
# calculate accuracy according to Olofsson's method
area_acc <- sits_accuracy(label_cube, validation_csv = ground_truth)
# print the area estimated accuracy
area_acc
```

```
#> Area Weigthed Statistics
#> Overall Accuracy = 0.76
#>
#> Area-Weighted Users and Producers Accuracy
#>       User Producer
#> Cerrado 1.00 0.67
#> Forest 0.60  1.00
#> Pasture 0.75 0.61
#> Soy_Corn 0.86 0.72
#>
#> Mapped Area x Estimated Area (ha)
#>       Mapped Area (ha) Error-Adjusted Area (ha)
```

```
#> Cerrado     32568.99           48735.95
#> Forest      80834.84           48500.90
#> Pasture     19029.43           23393.51
#> Soy_Corn    63850.02           75652.92
#>       Conf Interval (ha)
#> Cerrado       31687.26
#> Forest        38808.81
#> Pasture       20163.53
#> Soy_Corn      37558.62
```

This is an illustrative example to express the situation where there is a limited number of ground truth points. As a result of a limited validation sample, the estimated confidence interval in area estimation is large. This indicates a questionable result. We recommend that users follow the procedures recommended by [70] to estimate the number of ground truth measures per class that are required to get a reliable estimate.

Case studies

This chapter presents case studies using sits

```
period <- "P16D"

output_dir <- "/data/cubes_RO_20m_P16D_cloud"

start_date <- "2020-06-01"
end_date <- "2021-09-11"

bands <- c("B02", "B03", "B04", "B05", "B08", "B8A", "B11", "B12",
   "CLOUD")

tile_names <- c("19LHH", "19LHJ", "20LKM", "20LKN", "20LKP",
   "20LLM", "20LLN", "20LLP", "20LLQ", "20LLR", "20LMM", "20LMN",
   "20LMP", "20LMQ", "20LMR", "20LNL", "20LNM", "20LNN", "20LNP",
   "20LNQ", "20LNR", "20LPL", "20LPM", "20LPN", "20LPP", "20LPQ",
   "20LQL", "20LQM", "20LQN", "20LRN", "20LRM", "20LRL", "20LQK",
   "20LPR", "20MNS", "20MMS", "20LKQ", "19LHK", "19LGK", "20LKR")

#### source ####

# create output directory
if (!dir.exists(output_dir)) dir.create(output_dir)
stopifnot(dir.exists(output_dir))


s2_cube_2021 <- sits_cube(source = "AWS", collection = "SENTINEL-S2-L2A-
    COGS",
   bands = bands, tiles = tile_names, start_date = start_date,
   end_date = end_date, multicores = 20)
```

```
# regularize
system.time({
   reg_cube <- sits_regularize(cube = s2_cube_2021, period = period,
      res = 20, output_dir = output_dir, multicores = 5, multithreads =
          1,
      memsize = 120)
})

sits_apply(local, NBR = (B08 - B12)/(B08 + B12), memsize = 256,
   multicores = 72, output_dir = cube_dir)

sits_apply(local, EVI = 2.5 * (B08 - B04)/((B08 + 6 * B04 - 7.5 *
   B02) + 1), memsize = 256, multicores = 72, output_dir = cube_dir)

cube_dir <- "/data/cubes_RO_20m_P16D_cloud"
out_dir <- "/data/results/RO_P16D_cloud/tcnn"
rds_samples_file <- "/data/results/RO_P16D_cloud/tcnn/samples_tb.rds"
rds_model_file <- "/data/results/RO_P16D_cloud/tcnn/model.rds"
in_samples_file <- "/home/rolf.simoes/data/samples_RO/
    Amostras_RO_sits_names.csv"


# --- source ----

library(dplyr)
library(sits)

if (!dir.exists(out_dir)) dir.create(out_dir, recursive = TRUE)
stopifnot(dir.exists(out_dir))

stopifnot(file.exists(in_samples_file))

local <- sits_cube(source = "AWS", collection = "SENTINEL-S2-L2A-COGS",
   data_dir = cube_dir, parse_info = c("x", "tile", "band",
      "date"), multicores = 64)

if (!file.exists(rds_samples_file)) {

   samples_tb <- sits_get_data(cube = local, file = in_samples_file,
      multicores = 10)

   saveRDS(samples_tb, file = rds_samples_file)
} else {
   samples_tb <- readRDS(rds_samples_file)
```

```
}

if (!file.exists(rds_model_file)) {

   model <- sits_train(data = samples_tb, ml_method = sits_lightgbm())

   saveRDS(model, file = rds_model_file)
} else {
   model <- readRDS(rds_model_file)
}


probs <- sits_classify(data = local, ml_model = model, output_dir =
    out_dir,
   memsize = 400, multicores = 100, progress = TRUE)


sits:::.sits_parallel_start(10, FALSE)

my_libpaths <- .libPaths()

parallel::clusterExport(sits:::sits_env$cluster, c("probs", "out_dir",
   "my_libpaths"))

parallel::clusterEvalQ(sits:::sits_env$cluster, .libPaths(my_libpaths))

parallel::clusterEvalQ(sits:::sits_env$cluster, library(sits))

probs_bayes_lst <- sits:::.sits_parallel_map(seq_len(nrow(probs)),
   function(i) {

      probs_tile <- probs[i, ]

      sits_smooth(cube = probs_tile, type = "bayes", window_size = 5,
         smoothenes = 20, covar = FALSE, multicores = 1, memsize = 48,
         output_dir = out_dir)
   })

probs_bayes <- dplyr::bind_rows(probs_bayes_lst)

parallel::clusterExport(sits:::sits_env$cluster, c("probs_bayes"))

class_lst <- sits:::.sits_parallel_map(seq_len(nrow(probs_bayes)),
   function(i) {
```

```
      probs_bayes_tile <- probs_bayes[i, ]

      sits_label_classification(cube = probs_bayes_tile, multicores = 1,
         memsize = 48, output_dir = out_dir)
   })

class <- dplyr::bind_rows(class_lst)


sits:::.sits_parallel_stop()
```

Design and extensibility considerations

This chapter presents design decision for the **sits** package and shows how users can add their own machine learning algorithms to work with sits.

Design decisions

Compared with existing tools, sits has distinctive features:

1. A consistent API that encapsulates the entire land classification workflow in a few commands.
2. Integration with data cubes and Earth observation image collections available in cloud services such as AWS and Microsoft.
3. A single interface for different machine learning and deep learning algorithms.
4. Internal support for parallel processing, without requiring users to learn how to improve the performance of their scripts.
5. Support for efficient processing of large areas in a user-transparent way.
6. Innovative methods for sample quality control and post-processing.
7. Capacity to run on virtual machines in cloud environments.

Considering the aims and design of **sits**, it is relevant to discuss how its design and implementation choices differ from other software for big EO data analytics, such as Google Earth Engine [71], Open Data Cube [72] and openEO [73]. In what follows, we compare **sits** to each of these solutions.

Google Earth Engine (GEE) [71] uses the Google distributed file system [74] and its map-reduce paradigm [75]. By combining a flexible API with an efficient back-end processing, GEE has become a widely used platform [76]. However, GEE is restricted to

the Google environment and does not provide direct support for deep learning. By contrast, **sits** aims to support different cloud environments and to allow advances in data analysis by providing a user-extensible interface to include new machine learning algorithms.

The Open Data Cube (ODC) is an important contribution to the EO community and has proven its usefulness in many domains [77]. It reads subsets of image collections and makes them available to users as a Python `xarray` structure. ODC does not provide an API to work with `xarrays`, relying on the tools available in Python. This choice allows much flexibility at the cost of increasing the learning curve. It also means that temporal continuity is restricted to the `xarray` memory data structure; cases where tiles from an image collection have different timelines are not handled by ODC. The design of **sits** takes a different approach, favouring a simple API with few commands to reduce the learning curve. Processing and handling large image collections in **sits** does not require knowledge of parallel programming tools. Thus, **sits** and ODC have different aims and will appeal to different classes of users.

Designers of the openEO API [73] aim to support applications that are both language-independent and server-independent. To achieve their goals, openEO designers use microservices based on REST protocols. The main abstraction of openEO is a *process*, defined as an operation that performs a specific task. Processes are described in JSON and can be chained in process graphs. The software relies on server-specific implementations that translate an openEO process graph into an executable script. Arguably, openEO is the most ambitious solution for reproducibility across different EO data cubes. To achieve its goals, openEO needs to overcome some challenges. Most data analysis functions are not self-contained. For example, machine learning algorithms depend on libraries such as TensorFlow and Torch. If these libraries are not available in the target environment, the user-requested process may not be executable. Thus, while the authors expect openEO to evolve into a widely-used API, it is not yet feasible to base an user-driven operational software such as **sits** in openEO.

Designing software for big Earth observation data analysis requires making compromises between flexibility, interoperability, efficiency, and ease of use. GEE is constrained by the Google environment and excels at certain tasks (e.g., pixel-based processing) while being limited at others such as deep learning. ODC allows users complete flexibility in the Python ecosystem, at the cost of limitations when working with large areas and requiring programming skills. The openEO API achieves platform independence but needs additional effort in designing drivers for specific languages and cloud services. While the **sits** API provides a simple and powerful environment for land classification, it has currently no support for other kinds of EO applications. Therefore, each of these solutions has benefits and drawbacks. Potential users need to understand the design choices and constraints to decide which software best meets their needs.

[1] C. E. Woodcock, T. R. Loveland, M. Herold, and M. E. Bauer, "Transitioning from change detection to monitoring with remote sensing: A paradigm shift," *Remote Sensing of Environment*, vol. 238, p. 111558, 2020, doi: 10.1016/j.rse.2019.111558[15].

[2] M. Appel and E. Pebesma, "On-Demand Processing of Data Cubes from Satellite Image Collections with the gdalcubes Library," *Data*, vol. 4, no. 3, pp. 1–16, 2019, doi: 10.3390/data4030092[16].

[3] K. R. Ferreira *et al.*, "Earth Observation Data Cubes for Brazil: Requirements, Methodology and Products," *Remote Sensing*, vol. 12, no. 24, p. 4033, 2020, doi: 10.3390/rs12244033[17].

[4] H. Wickham and G. Grolemund, *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc., 2017.

[5] E. F. Lambin and M. Linderman, "Time series of remote sensing data for land change science," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 44, no. 7, pp. 1926–1928, 2006.

[6] P. M. Atkinson, C. Jeganathan, J. Dash, and C. Atzberger, "Inter-comparison of four models for smoothing satellite sensor time-series data to estimate vegetation phenology," *Remote Sensing of Environment*, vol. 123, pp. 400–417, 2012.

[7] J. Zhou, L. Jia, M. Menenti, and B. Gorte, "On the performance of remote sensing time series reconstruction methods: A spatial comparison," *Remote Sensing of Environment*, vol. 187, pp. 367–384, 2016.

[8] J. Chen, Per. Jönsson, M. Tamura, Z. Gu, B. Matsushita, and L. Eklundh, "A simple method for reconstructing a high-quality NDVI time-series data set based on the Savitzky–Golay filter," *Remote Sensing of Environment*, vol. 91, no. 3, pp. 332–344, 2004, doi: 10.1016/j.rse.2004.03.014[18].

[9] C. Atzberger and P. H. Eilers, "Evaluating the effectiveness of smoothing algorithms in the absence of ground reference measurements," *International Journal of Remote Sensing*, vol. 32, no. 13, pp. 3689–3709, 2011.

[10] A. E. Maxwell, T. A. Warner, and F. Fang, "Implementation of machine-learning classification in remote sensing: An applied review," *International Journal of Remote Sensing*, vol. 39, no. 9, pp. 2784–2817, 2018.

[15] https://doi.org/10.1016/j.rse.2019.111558
[16] https://doi.org/10.3390/data4030092
[17] https://doi.org/10.3390/rs12244033
[18] https://doi.org/10.1016/j.rse.2004.03.014

[11] B. Frenay and M. Verleysen, "Classification in the Presence of Label Noise: A Survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 5, pp. 845–869, 2014, doi: 10.1109/TNNLS.2013.2292894[19].

[12] E. Keogh, J. Lin, and W. Truppel, "Clustering of time series subsequences is meaningless: Implications for previous and future research," in *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, 2003, pp. 115–122.

[13] F. Petitjean, J. Inglada, and P. Gancarski, "Satellite Image Time Series Analysis Under Time Warping," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 50, no. 8, pp. 3081–3095, 2012, doi: 10.1109/TGRS.2011.2179050[20].

[14] V. Maus, G. Camara, R. Cartaxo, A. Sanchez, F. M. Ramos, and G. R. Queiroz, "A Time-Weighted Dynamic Time Warping Method for Land-Use and Land-Cover Mapping," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 8, pp. 3729–3739, 2016, doi: 10.1109/JSTARS.2016.2517118[21].

[15] J. H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[16] W. M. Rand, "Objective Criteria for the Evaluation of Clustering Methods," *Journal of the American Statistical Association*, vol. 66, no. 336, pp. 846–850, 1971, doi: 10.1080/01621459.1971.10482356[22].

[17] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990, doi: 10.1109/5.58325[23].

[18] L. A. Santos, K. R. Ferreira, G. Camara, M. C. A. Picoli, and R. E. Simoes, "Quality control and class noise reduction of satellite image time series," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 177, pp. 75–88, 2021, doi: 10.1016/j.isprsjprs.2021.04.014[24].

[19] R. Wehrens and J. Kruisselbrink, "Flexible Self-Organizing Maps in kohonen 3.0," *Journal of Statistical Software*, vol. 87, no. 1, pp. 1–18, 2018, doi: 10.18637/jss.v087.i07[25].

[19] https://doi.org/10.1109/TNNLS.2013.2292894
[20] https://doi.org/10.1109/TGRS.2011.2179050
[21] https://doi.org/10.1109/JSTARS.2016.2517118
[22] https://doi.org/10.1080/01621459.1971.10482356
[23] https://doi.org/10.1109/5.58325
[24] https://doi.org/10.1016/j.isprsjprs.2021.04.014
[25] https://doi.org/10.18637/jss.v087.i07

[20] L. A. Santos, K. Ferreira, M. Picoli, G. Camara, R. Zurita-Milla, and E.-W. Augustijn, "Identifying Spatiotemporal Patterns in Land Use and Cover Samples from Satellite Image Time Series," *Remote Sensing*, vol. 13, no. 5, p. 974, 2021, doi: 10.3390/rs13050974[26].

[21] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, p. 27, 2019, doi: 10.1186/s40537-019-0192-5[27].

[22] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, 2002.

[23] K. Janowicz, S. Scheider, T. Pehle, and G. Hart, "Geospatial semantics and linked spatiotemporal data – Past, present, and future," *Semantic Web*, vol. 3, no. 4, pp. 321–332, 2012, doi: 10.3233/SW-2012-0077[28].

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[25] M. Belgiu and L. Dragut, "Random Forest in remote sensing: A review of applications and future directions," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 114, pp. 24–31, 2016.

[26] G. Mountrakis, J. Im, and C. Ogole, "Support vector machines in remote sensing: A review," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 66, no. 3, pp. 247–259, 2011.

[27] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794, doi: 10.1145/2939672.2939785[29].

[28] L. Parente, E. Taquary, A. P. Silva, C. Souza, and L. Ferreira, "Next Generation Mapping: Combining Deep Learning, Cloud Computing, and Big Remote Sensing Data," *Remote Sensing*, vol. 11, no. 23, p. 2881, 2019, doi: 10.3390/rs11232881[30].

[29] C. Pelletier, G. I. Webb, and F. Petitjean, "Temporal Convolutional Neural Network for the Classification of Satellite Image Time Series," *Remote Sensing*, vol. 11, no. 5, 2019.

[30] H. Fawaz *et al.*, "InceptionTime: Finding AlexNet for time series classification," *Data Mining and Knowledge Discovery*, vol. 34, no. 6, pp. 1936–1962, 2020, doi: 10.1007/s10618-020-00710-y[31].

[26] https://doi.org/10.3390/rs13050974
[27] https://doi.org/10.1186/s40537-019-0192-5
[28] https://doi.org/10.3233/SW-2012-0077
[29] https://doi.org/10.1145/2939672.2939785
[30] https://doi.org/10.3390/rs11232881
[31] https://doi.org/10.1007/s10618-020-00710-y

[31] V. Garnot, L. Landrieu, S. Giordano, and N. Chehata, "Satellite Image Time Series Classification With Pixel-Set Encoders and Temporal Self-Attention," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 12322–12331, doi: 10.1109/CVPR42600.2020.01234[32].

[32] M. Rußwurm, C. Pelletier, M. Zollner, S. Lefèvre, and M. Körner, "BreizhCrops: A Time Series Dataset for Crop Type Mapping," 2020.

[33] M. Picoli *et al.*, "Big earth observation time series analysis for monitoring Brazilian agriculture," *ISPRS journal of photogrammetry and remote sensing*, vol. 145, pp. 328–339, 2018, doi: 10.1016/j.isprsjprs.2018.08.007[33].

[34] M. C. A. Picoli *et al.*, "CBERS data cube: A powerful technology for mapping and monitoring Brazilian biomes." in *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2020, vol. V–3–2020, pp. 533–539, doi: 10.5194/isprs-annals-V-3-2020-533-2020[34].

[35] R. Simoes *et al.*, "Land use and cover maps for Mato Grosso State in Brazil from 2001 to 2017," *Scientific Data*, vol. 7, no. 1, p. 34, 2020, doi: 10.1038/s41597-020-0371-4[35].

[36] V. Maus, G. Câmara, M. Appel, and E. Pebesma, "dtwSat: Time-Weighted Dynamic Time Warping for Satellite Image Time Series Analysis in R," *Journal of Statistical Software*, vol. 88, no. 5, pp. 1–31, 2019, doi: 10.18637/jss.v088.i05[36].

[37] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. New York, EUA: Springer, 2013.

[38] J. S. Wright *et al.*, "Rainforest-initiated wet season onset over the southern Amazon," *Proceedings of the National Academy of Sciences*, 2017, doi: 10.1073/pnas.1621516114[37].

[39] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. New York: Springer, 2009.

[40] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[41] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[42] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[32] https://doi.org/10.1109/CVPR42600.2020.01234
[33] https://doi.org/10.1016/j.isprsjprs.2018.08.007
[34] https://doi.org/10.5194/isprs-annals-V-3-2020-533-2020
[35] https://doi.org/10.1038/s41597-020-0371-4
[36] https://doi.org/10.18637/jss.v088.i05
[37] https://doi.org/10.1073/pnas.1621516114

[43] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[44] Z. Wang, W. Yan, and T. Oates, "Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline," 2017.

[45] M. Rußwurm and M. Korner, "Temporal vegetation modelling using long short-term memory networks for crop identification from medium-resolution multi-spectral satellite images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 11–19.

[46] M. Russwurm and M. Korner, "Multi-temporal land cover classification with sequential recurrent encoders," *ISPRS International Journal of Geo-Information*, vol. 7, no. 4, p. 129, 2018.

[47] R. Simoes *et al.*, "Satellite Image Time Series Analysis for Big Earth Observation Data," *Remote Sensing*, vol. 13, no. 13, p. 2428, 2021, doi: 10.3390/rs13132428[38].

[48] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90[39].

[49] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 2, pp. 107–116, 1998, doi: 10.1142/S0218488598000094[40].

[50] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: A review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[51] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, 2017, vol. 30.

[52] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." arXiv, 2019, doi: 10.48550/arXiv.1810.04805[41].

[53] T. B. Brown *et al.*, "Language Models are Few-Shot Learners." arXiv, 2020, doi: 10.48550/arXiv.2005.14165[42].

[38] https://doi.org/10.3390/rs13132428
[39] https://doi.org/10.1109/CVPR.2016.90
[40] https://doi.org/10.1142/S0218488598000094
[41] https://doi.org/10.48550/arXiv.1810.04805
[42] https://doi.org/10.48550/arXiv.2005.14165

[54] V. S. F. Garnot and L. Landrieu, "Lightweight Temporal Self-attention for Classifying Satellite Images Time Series," in *Advanced Analytics and Learning on Temporal Data*, 2020, pp. 171–181, doi: 10.1007/978-3-030-65742-0_12[43].

[55] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *arXiv:1206.5533 [cs]*, 2012.

[56] R. M. Schmidt, F. Schneider, and P. Hennig, "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers," in *Proceedings of the 38th International Conference on Machine Learning*, 2021, pp. 9367–9376.

[57] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012.

[58] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018, doi: 10.1137/16M1080173[44].

[59] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization." arXiv, 2017, doi: 10.48550/arXiv.1412.6980[45].

[60] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," *arXiv:1711.05101 [cs, math]*, 2019.

[61] A. Defazio and S. Jelassi, "Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization." arXiv, 2021, doi: 10.48550/arXiv.2101.11075[46].

[62] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive Methods for Nonconvex Optimization," in *Advances in Neural Information Processing Systems*, 2018, vol. 31.

[63] X. Huang, Q. Lu, L. Zhang, and A. Plaza, "New postprocessing methods for remote sensing image classification: A systematic study," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 52, no. 11, pp. 7140–7159, 2014.

[64] K. Schindler, "An overview and comparison of smooth labeling methods for land-cover classification," *IEEE transactions on geoscience and remote sensing*, vol. 50, no. 11, pp. 4534–4545, 2012.

[65] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, 1998, pp. 839–846, doi: 10.1109/ICCV.1998.710815[47].

[43] https://doi.org/10.1007/978-3-030-65742-0_12
[44] https://doi.org/10.1137/16M1080173
[45] https://doi.org/10.48550/arXiv.1412.6980
[46] https://doi.org/10.48550/arXiv.2101.11075
[47] https://doi.org/10.1109/ICCV.1998.710815

[66] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "A gentle introduction to bilateral filtering and its applications," in *ACM SIGGRAPH 2007 courses*, 2007, pp. 1–es, doi: 10.1145/1281500.1281602[48].

[67] A. D'Amour *et al.*, "Underspecification Presents Challenges for Credibility in Modern Machine Learning," *arXiv:2011.03395 [cs, stat]*, 2020.

[68] A. M. J.-C. Wadoux, G. B. M. Heuvelink, S. de Bruin, and D. J. Brus, "Spatial cross-validation is not the right way to evaluate map accuracy," *Ecological Modelling*, vol. 457, p. 109692, 2021, doi: 10.1016/j.ecolmodel.2021.109692[49].

[69] P. Olofsson, G. M. Foody, S. V. Stehman, and C. E. Woodcock, "Making better use of accuracy data in land change studies: Estimating accuracy and area and quantifying uncertainty using stratified estimation," *Remote Sensing of Environment*, vol. 129, pp. 122–131, 2013, doi: 10.1016/j.rse.2012.10.031[50].

[70] P. Olofsson, G. M. Foody, M. Herold, S. V. Stehman, C. E. Woodcock, and M. A. Wulder, "Good practices for estimating area and assessing accuracy of land change," *Remote Sensing of Environment*, vol. 148, pp. 42–57, 2014.

[71] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, "Google Earth Engine: Planetary-scale geospatial analysis for everyone," *Remote Sensing of Environment*, vol. 202, pp. 18–27, 2017.

[72] A. Lewis *et al.*, "The Australian Geoscience Data Cube — Foundations and lessons learned," *Remote Sensing of Environment*, vol. 202, pp. 276–292, 2017, doi: 10.1016/j.rse.2017.03.015[51].

[73] M. Schramm *et al.*, "The openEO API–Harmonising the Use of Earth Observation Cloud Services Using Virtual Data Cube Functionalities," *Remote Sensing*, vol. 13, no. 6, p. 1125, 2021, doi: 10.3390/rs13061125[52].

[74] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43, doi: 10.1145/945445.945450[53].

[75] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008, doi: 10.1145/1327452.1327492[54].

[48] https://doi.org/10.1145/1281500.1281602
[49] https://doi.org/10.1016/j.ecolmodel.2021.109692
[50] https://doi.org/10.1016/j.rse.2012.10.031
[51] https://doi.org/10.1016/j.rse.2017.03.015
[52] https://doi.org/10.3390/rs13061125
[53] https://doi.org/10.1145/945445.945450
[54] https://doi.org/10.1145/1327452.1327492

[76] M. Amani *et al.*, “Google Earth Engine Cloud Computing Platform for Remote Sensing Big Data Applications: A Comprehensive Review,” *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 13, pp. 5326–5350, 2020, doi: 10.1109/JSTARS.2020.3021052[55].

[77] G. Giuliani, B. Chatenoux, T. Piller, F. Moser, and P. Lacroix, “Data Cube on Demand (DCoD): Generating an earth observation Data Cube anywhere in the world,” *International Journal of Applied Earth Observation and Geoinformation*, vol. 87, p. 102035, 2020, doi: 10.1016/j.jag.2019.102035[56].

[55] https://doi.org/10.1109/JSTARS.2020.3021052
[56] https://doi.org/10.1016/j.jag.2019.102035