sits: Data Analysis and Machine Learning on Earth Observation Data Cubes with Satellite Image Time Series

Rolf Simoes Gilberto Camara Felipe Souza
Lorena Santos Pedro R. Andrade Charlotte Peletier
Alexandre Carvalho Karine Ferreira Gilberto Queiroz
Victor Maus

2021-09-27

Contents

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 since our growing demand for natural resources has caused significant environmental impacts.

This book presents **sits**, an open-source R package for land use and land cover change mapping using satellite image time series. The package uses machine learning techniques to classify image time series obtained from data cubes. Methods available include linear and quadratic discrimination analysis, support vector machines, random forests, boosting, deep learning, and convolutional neural networks. The package also provides functions for post-processing and sample quality assessment.

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 are new to R, we highly recommend the excellent self-learning book "Hands-On Programming with R" by Garrett Golemund from RStudio. If you want more information on the data science methods used in the book, please look at the following references:

- Wickham, H.; Golemund, G., "R for Data Science". O'Reilly, 2017.

- James, G.; Witten, D.; Hastie, T.; Tibshirani, R. "An Introduction to Statistical Learning with Applications in R". Springer, 2013.

- Goodfellow, I; Bengio, Y.; Courville, A. "Deep Learning". MIT Press, 2016.

How to use this book

This book describes **sits** version 0.13.1. Download and install the package as explained in the Setup. 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	Provides an overview of **sits**.
Chr 2	Describes how to work with Earth observation data cubes.
Chr 3	Describes how to access information from time series.
Chr 4	Improving the quality of the samples used in training models
Chr 5	Presents the machine learning techniques available in sits.
Chr 6	Describes how to classify satellite images associated with Earth observation data cubes.
Chr 7	Describes smoothing methods to reclassify the pixels based on the machine learning probabilities
Chr 8	Presents the validation and accuracy measures.
Chr 9	Presents case studies of LUCC classification.
Chr 10	How to develop extensions to **sits**.

Main reference for sits

if you use sits, 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.

Publications using sits

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

2019

- Alber Sanchez, Michelle Picoli, et al., "Land Cover Classifications of Clear-cut Deforestation Using Deep Learning." In: SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA (GEOINFO), 2019, São José dos Campos. São José dos Campos: INPE, 2019. On-line.

- Lorena Santos, Karine Ferreira, et al., "Self-Organizing Maps in Earth Observation Data Cubes Analysis." 13th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM+ 2019), Barcelona, Spain, June 26-28, 2019.

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.

Reproducible papers used in building sits functions

We thank the authors of these papers for making their code available to be used in sits.

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

- [2] 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.
- [3] Pebesma, Edzer, "Simple Features for R: Standardized Support for Spatial Vector Data." R Journal, 10(1):2018.
- [4] Pelletier, Charlotte, Geoffrey I. Webb, and Francois Petitjean. "Temporal Convolutional Neural Network for the Classification of Satellite Image Time Series." Remote Sensing 11 (5), 2019.
- [5] Wehrens, Ron and Kruisselbrink, Johannes. "Flexible Self-Organising Maps in kohonen 3.0." Journal of Statistical Software, 87, 7 (2018).

Setup

sits is currently available on GitHub. Thus, installing the package can be accomplished via devtools, as presented by the code snippet below:

```
devtools::install_github("e-sensing/sits", dependencies = TRUE)
```

Please also install the "sitsdata" package, which contains data for the examples in the book.

```
devtools::install_github("e-sensing/sitsdata")
```

```
<!--chapter:end:01-setup.Rmd-->

# Acknowledgements {-}

The authors would like to thank all the researchers that provided data samples used in the exampl

This work was partially funded by the São Paulo Research Foundation (FAPESP) through the eScience

This work has also been supported by the International Climate Initiative of the Germany Federal

The authors would like to acknowledge the contributions of Marius Appel, Tim Appelhans, Henrik Be

<!--chapter:end:02-acknowledgements.Rmd-->

# (PART) Overview {-}

# A taste of sits
```

This chapter present an overview of `sits` by showing an application example. For detailed description of the functions, please see the following chapters.

Earth observation (EO) satellites provide a common and consistent set of information about the planet's land and oceans. Recently, most space agencies have adopted open data policies, making unprecedented amounts of satellite data available for research and operational use. This data deluge has brought about a significant challenge: *How to design and build technologies that allow the Earth observation community to analyze big data sets?*

In this book, we present `sits`, an open-source R package for satellite image time series analysis. It provides support on how to use machine learning techniques with image time series. The package supports the complete cycle of data analysis for time series classification, including data acquisition, visualization, filtering, clustering, classification, validation, and post-processing.

The package adopts a *time-first, space-later* approach, where each spatial location is associated to a time series. A set of locations with known labels is used to train a machine learning classifiers. The resulting model is applied to the data cube and each time series is classified separately. After the classification, spatial smoothing methods capture information from neighbors. This approach is illustrated in Figure 1.

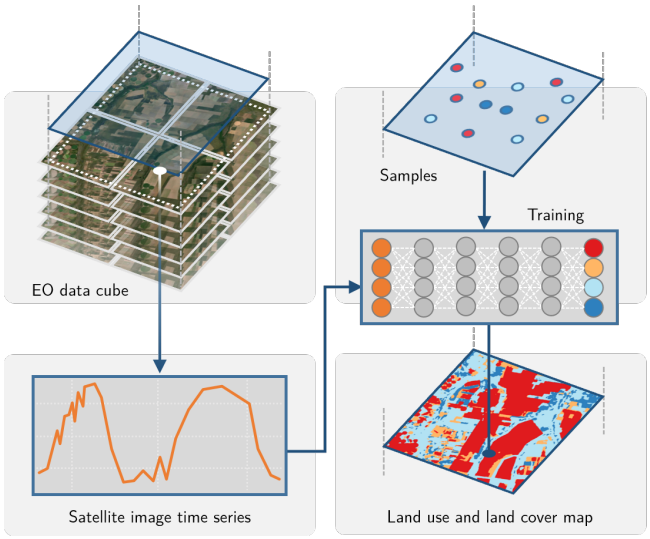


Figure 1: Using time series for land classification (source: authors)

0.1 Creating a Data Cube

In what follows, we introduce `sits` by showing a simple example fo land use and land cover classification. The first step is creating a data cube. The data cube is a set of analysis-ready MODIS MOD13Q1 images (collection 6) for the

Sinop region of the State of Mato Grosso, Brazil, in the bands "NDVI" and "EVI," covering a one-year period from 2013-09-14. (We use the convention "year-month-day" for dates). Each band has 23 instances, each covering a 16-day period, which is the standard for the MOD13Q1 product[1]. The data is available in the R package `sitsdata`, which contains data for the examples in this book. The code below shows how to create data cubes from local files. The data has been obtained from the Brazil Data Cube (BDC) repository and downloaded to be included in the `sitsdata` package. To work directly with data cubes in repositories such as AWS, MS/AZURE and the Brazil Data Cube, please see Chapter 2.

When building data cubes from images stored in a local machine, users need to provide information about the original source from with the data was downloaded. The reason to included such information is because there are no standards for the metadata used to process an image. Unfortunately, information such as band names, maximum/minimum values, and cloud pixels are set by the provider. When `sits` accesses local data, it needs to know where the data comes from. Hence the need to include `origin` and `collection` parameters when defining a local cube.

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

# create a raster metadata file based on the information about the files
sinop_cube <- sits_cube(source = "LOCAL",
                    origin = "BDC",
                    collection  = "MOD13Q1-6",
                    name = "Sinop",
                    data_dir = data_dir,
                    parse_info = c("X1", "X2", "tile", "band", "date")
)
# plot the EVI band for the first date (2013-09-14)
plot(sinop_cube, band = "EVI", time = 1)
```

```
#> downsample set to c(1)
```

The `sits_cube()` function defines a *data cube*, which is an organized collection of images covering a geographical area in a given time interval. Data cubes can be conceived as a 3D array of pixels, where each pixel is associated to a time series. All pixels share the same timeline and the same set of attributes (usually spectral bands). When a data cube is defined, the values of the images are not loaded in memory. The output of `sits_cube()` is a table which contains the metadata that describes the actual image data. For more details on how to define and work with data cubes, please see Chapter 2.

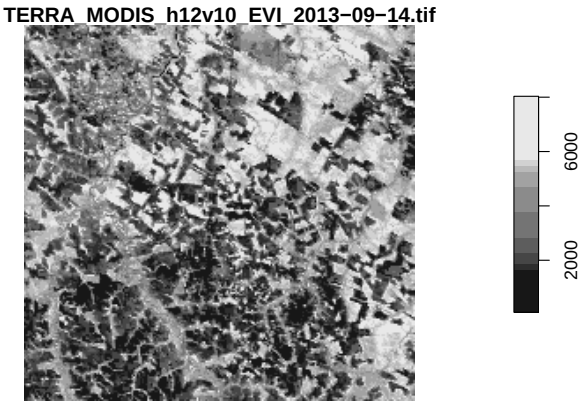


Figure 2: EVI band for 2013-09-14

0.2 The time series table

To classify all of the time series associated to a data cube, `sits` uses machine learning models. To train the models, `sits` uses a tabular data structure that stores individual time series. 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[1:3,]
```

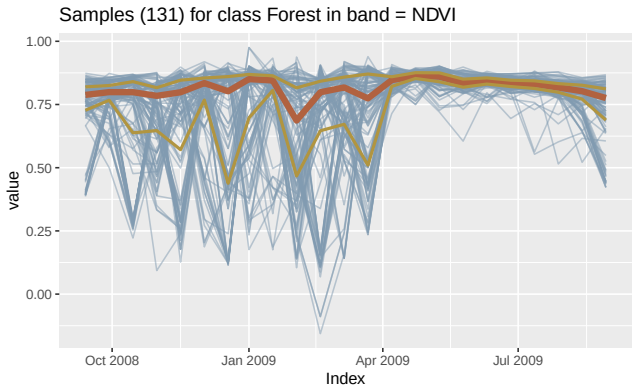
```
#> # A tibble: 3 x 7
#>   longitude latitude start_date end_date   label   cube    time_series
#>       <dbl>    <dbl> <date>     <date>     <chr>   <chr>   <list>
#> 1     -55.2   -10.8  2013-09-14 2014-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 2     -57.8    -9.76 2006-09-14 2007-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 3     -51.9   -13.4  2014-09-14 2015-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
```

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 Chapter 3.

It is useful to visualize the dispersion of the time series. In what follows, for

brevity we will select only one label ("Forest") and one index ("EVI") to show. 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)
```



0.3 Training a machine learning model

After obtaining the time series, the next step is to select a suitable subset to use as training samples for a machine learning model. In this case, the time series data has four attributes ("EVI," "NDVI," "NIR," "MIR") and the data cube is composed only with data from the "NDVI" and "EVI" indexes. We extract the "NDVI" and "EVI" indexes from the time series data set and use the resulting data for training a model. To build the classification model, we have chosen `sits_TempCNN()` from the methods available. This method implements a 1D convolution neural network [2]. After training the model, we plot the result to how well it has converged to match the input data. For more details on the machine learning methods please see Chapter 5.

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

#select a tempCNN model
tcnn_model <- sits_train(data = samples_2bands,
                         ml_method = sits_TempCNN())
plot(tcnn_model)
```

```
#> `geom_smooth()` using formula 'y ~ x'
```

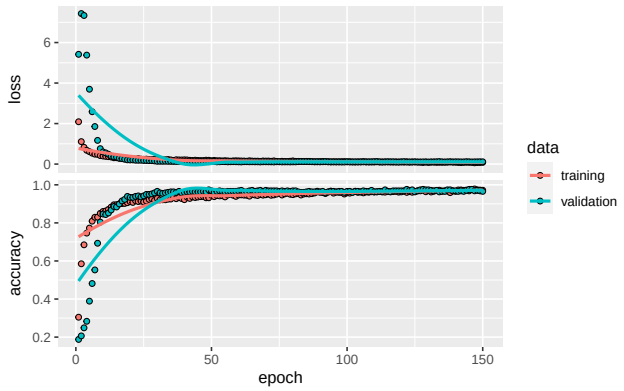


Figure 3: Validation of TempCNN model

0.4 Data cube classification

The next step is to classify the data cube. This is achieved by using the `sits_classify()` function. The classification produces a set of probability maps, one for each class. For each map, the value of a pixel is proportional to the the probability that it belongs to the class. To visualize the result, we plot the probability maps. In the example below, we show the maps of two classes ("Forest" and "Pasture"). Details of the classification process are available in Chapter 6.

```
# classify the raster image
sinop_probs <- sits_classify(data = sinop_cube,
                             ml_model = tcnn_model)

plot(sinop_probs, labels = c("Forest", "Pasture"))
```

```
#> downsample set to c(1,1,1)
```

0.5 Spatial smoothing

When working with big EO data sets, there is a considerable degree of data variability in each class. As a result, some pixels will be misclassified. These errors are more likely to occur in transition areas between classes or when dealing with mixed pixels. To offset these problems, `sits` includes a post-processing smoothing method based on Bayesian probability. The `sits_smooth()` function uses information from a pixel's neighborhood to reduce uncertainty about its label, which is illustrated below. After smoothing, we plot the probability maps for classes "Forest" and "Pasture" to compare with the previous plot. For more discussion on post-processing and smoothing methods, please see Chapter 7.

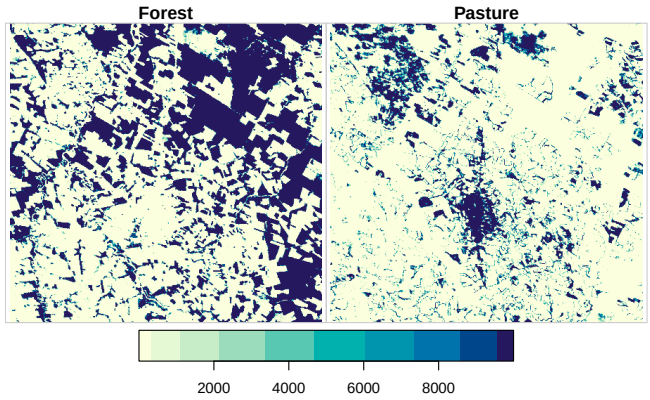


Figure 4: Probability map for classes of each pixel

```
# perform spatial smoothing
sinop_smooth <- sits_smooth(sinop_probs)
plot(sinop_smooth, labels = c("Forest", "Pasture"))
```

```
#> downsample set to c(1,1,1)
```

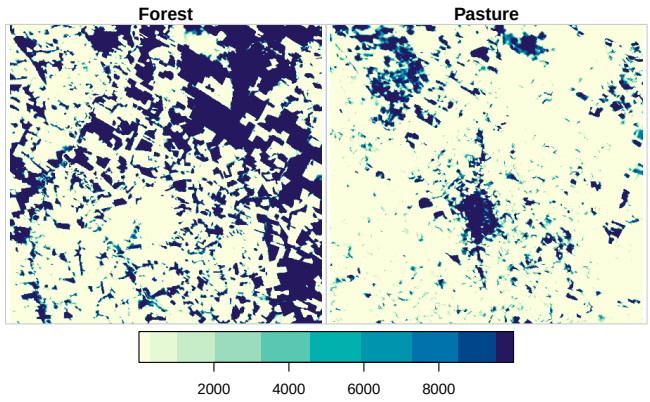


Figure 5: Smoothed probability maps

0.6 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
# (by default selecting the class with higher probability)
sinop_label <- sits_label_classification(sinop_smooth)
plot(sinop_label, title = "Sinop-label")
```

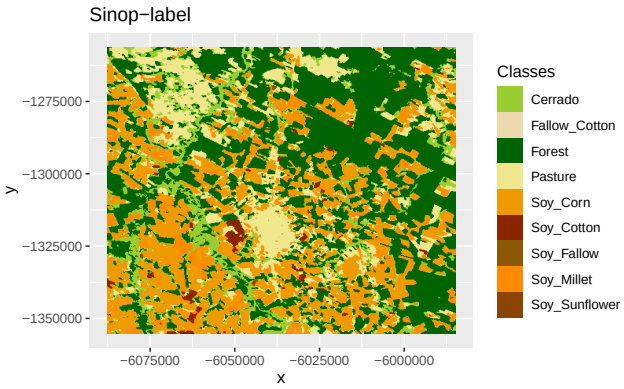


Figure 6: Labeled classification map

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

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

```
#> # A tibble: 1 x 4
#>   band              start_date end_date   path
#>   <chr>             <date>     <date>     <chr>
#> 1 Sinop_probs_class 2013-09-14 2014-08-29 /tmp/RtmpVvkeOW/Sinop_class_PROBS_bayes_v
```

0.7 How the sits API works

The core functions of the **sits** API are presented in Figure 7. Each function carries out one task of the land classification workflow. These functions are: (a) `sits_cube()` which creates a cube; (b) `sits_get_data()` which extracts training data from the cube; (c) `sits_train()` that trains a machine learning model; (d) `sits_classify()` which classifies the cube; (e) `sits_smooth()` that does the spatial smoothing; and (f) `sits_label_classication()` that produces the final labelled image. These six functions encapsulate the core of the package. Each of these core functions is described in a chapter in this book.

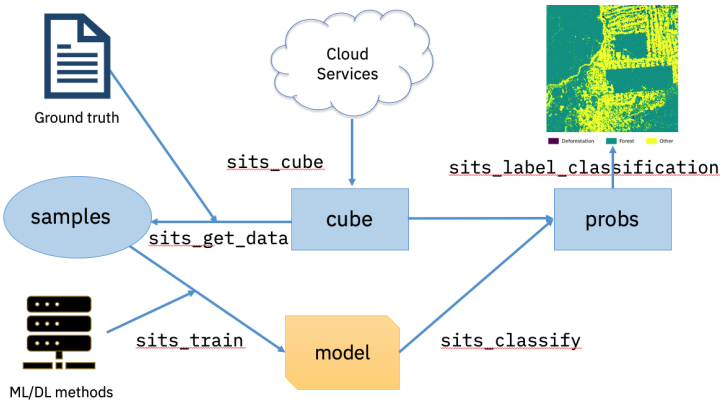


Figure 7: Main functions of the SITS API

0.8 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.

Chapter 1

Earth observation data cubes

This chapter describes how to use Earth observation data cubes in SITS.

1.1 Image data cubes as the basis for big Earth observation data analysis

Given the large sizes of the collections of Earth observation data available, there is a clear trend to use cloud computing. In this configuration, cloud services archive large satellite-generated data sets and provide computing facilities to process them. Users can share big Earth observation databases and minimize data download. Investment in infrastructure is minimized, and sharing of data and software increases.

To take full advantage of the cloud computing model, Earth observation data needs to be available to users as *data cubes*, whose aim is to organize satellite data for a given area in a consistent spatiotemporal arrangement. Generalizing [3], we consider that a *data cube* should meet the following definition:

1. A 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.

A data cube defines a regular partition of space-time. 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 data cube should provide a valid 2D image (see Figure 1.1).

Data cubes provide a useful abstraction for algorithms that extract information from big EO data sets. Machine learning and deep learning 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.

Currently, few cloud services support the above definition of data cubes. Cloud providers such as AWS and Microsoft organize their data as analysis-ready image collection. In the next section, we describe such collections and point out their differences to data cubes.

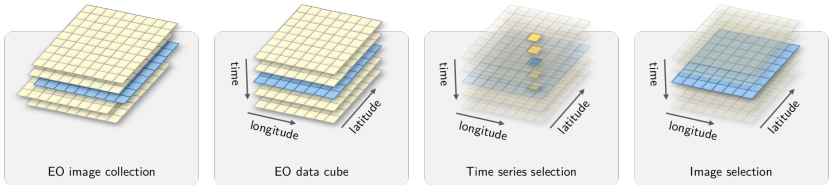


Figure 1.1: Conceptual view of data cubes (source: authors)

1.2 Analysis-ready data image collections

Data available in cloud services such as Amazon Web Service (AWS), Microsoft's Planetary Computer, and Digital Earth Africa (DE Africa) does not adhere to the definition of data cubes stated above. These data sets are better described as collections of analysis-ready data (ARD) which have been processed by space agencies to improve multidate comparability. Such processing includes conversion from radiance measures at the top of the atmosphere to reflectance measures from ground areas. Variations in sun incidence angles are also compensated. The image is usually reprocessed to a well-known cartographic projection. We define an *ARD image collections* as:

1. An ARD image collection is a set of files from a given sensor (or a combined set of sensors) that has been corrected to ensure comparability of measurements between different dates.
2. All images are reprojected to a cartographic projection following well-established standards.
3. Image collections are cropped into a tiling system.

4. In general, the timelines of the images that are part of one tile are not regular. Also, these timelines are not the same as those associated to a different tile.
5. ARD image collections do not guarantee that every pixel of an image has a valid value, since its images still contains cloudy or missing pixels.

Figure 1.2 shows images of tile "20LKP" of the Sentinel-2 Level 2A collection available in AWS for different dates. Some of images have a significant number of clouds. The values of cloudy pixels should be replaced by valid values before being ingested into a data cube.

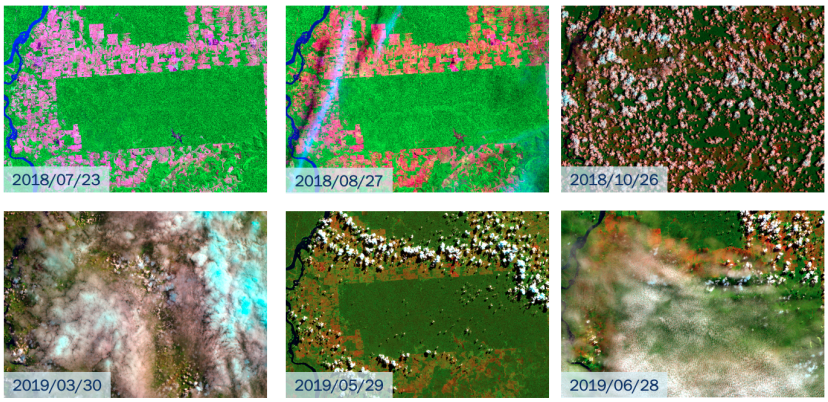


Figure 1.2: Sentinel-2 for tile 20LKP in different dates (source: authors).

A further point concerns the timeline of different tiles. Consider the neighboring Sentinel-2 tiles "20LLP" and "20LKP" for the period 2018-07-13 to 2019-07-28, as they are available in AWS. Tile 20LKP has 71 temporal instances and tile 20 LLP has 144 instances. To process large areas, all tiles have to be organized to follow the same timeline. Thus, users cannot rely on ARD image collections when working with large areas. Users that want to work on multiples tiles of an ARD image collection should use the `sits_regularize()` function to generate regularly spaced cubes in time from image collection. Please see the session "Regularizing data cubes" below.

1.3 Accessing Data Cubes and Image Collections in SITS

To obtain information on cloud image collection, `sits` uses information provided by implementations of the STAC (SpatioTemporal Asset Catalogue) protocol. [STAC] (https://stacspec.org/) is a specification of geospatial information which has been adopted by many large image collection providers (e.g., AWS, Microsoft, USGS). 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 R package.

All access to data cubes is done using `sits_cube()`. For cloud services such as AWS, DE Africa, Planetary Computer and the Brazil Data Cube (BDC), the user must provide similar parameters, as shown in the examples below. Please note that most of the examples in this chapter require access keys to cloud repositories and as such are not executable directly. Users need to provide the appropriate access keys to run the examples.

1.3.1 Accessing data cubes in Amazon Web Services

Users of Amazon Web Services (AWS) can access image collections available in the 'Earth on AWS' services using `sits`. For AWS, `sits` currently works with collection "sentinel-s2-l2a." This will be extended in later versions.

To work with AWS, users need to provide credentials using environment variables.

```
Sys.setenv(
    "AWS_ACCESS_KEY_ID"     = <your_access_key>,
    "AWS_SECRET_ACCESS_KEY" = <your_secret_access_key>,
    "AWS_DEFAULT_REGION"    = <your AWS region>,
    "AWS_ENDPOINT"          = <your AWS endpoint>,
    "AWS_REQUEST_PAYER"     = "requester"
)
```

Sentinel-2 level 2A files in AWS collection "sentinel-s2-l2a" are organized by sensor resolution. Bands "B02," "B03," "B04," and "B08" AWS are available in 10m resolution. Bands "B02," "B03," "B04," "B05," "B06," "BO7," B08","B8A","B11", and"B12" are available at 20m resolution. All 12 bands are available at 60m resolution. Because of the availability of some bands at different resolutions, users need to specify the `s2_resolution` parameter to create Sentinel-2 images data cubes in AWS using collection "sentinel-s2-l2a." In the example below, the user selects one Sentinel-2 tile. Each S2 tile is an 100x100 km2 orthoimage in UTM/WGS84 projection.

```
# creating a data cube in AWS
s2_cube <- sits_cube(source = "AWS",
                     name = "T20LKP_2018_2019",
                     collection = "sentinel-s2-l2a",
                     tiles = c("20LKP"),
                     start_date = as.Date("2018-07-18"),
                     end_date = as.Date("2018-07-23"),
                     bands = c("B02", "B03", "B04", "B08", "B11"),
                     s2_resolution = 20
)
```

The output of the `sits_cube()` function is composed of metadata about the

images that satisfy the requirements stated in its parameters (spatiotemporal extent, resolution, and area of interest). The `s2_cube` object created in the above statement is a tibble that has the information required for further processing, but does not contain the actual data.

Instead of specifying the region of interest by listing the image collection tiles, users can also provide a bounding box (`bbox`) whose parameters allow a selection of an area of interest. Bounding boxes can be defined using: (a) a named vector ("xmin," "ymin," "xmax," "ymax") with lat/long values in WGS 84; (b) an *sf* object from the *sf* package, a data frame with feature attributes and feature geometries; or (c) a GeoJSON geometry (RFC 7946). When selecting images that compose a data cube based on a `bbox`, sits does not crop them directly; the software selects the images that intersect with it. The information is used later by `sits_classify()`, when only the pixels inside the bounding box will be processed.

In SITS version 0.14.0, users who do not want to use the `sits_regularize()` function and prefer to use AWS images directly can only create valid data cubes if the images belong to the same tile. This limitation will be removed in future versions of the package.

1.3.2 Accessing the Brazil Data Cube

The Brazil Data Cube (BDC) is being developed by Brazil's National Institute for Space Research (INPE). Its goal is to create multidimensional data cubes of analysis-ready data Brazil. 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.

To access the Brazil Data Cube, users need to provide their credentials using environmental variables.

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

Creating a data cube using the BDC is similar to what is required for AWS. The user defines an image collection, a spatiotemporal extent, bands, and optionally a bounding box. In the example below, the data cube is defined as one tile ("022024") of "CB4_64_16D_STK-1" collection which holds CBERS AWFI

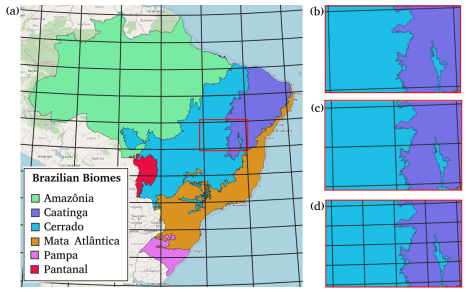


Figure 1.3: Hierarchical BDC tiling system showing overlayed on Brazilian Biomes (a), illustrating that one large tile (b) contains four medium tiles (c) and that medium tile contains four small tiles

images at 16 days resolution. Other collections include "LC8_30_16D_STK-1" (Landsat OLI images at 16 days), "S2-SEN2COR_10_16D_STK-1" (Sentinel-2 MSI images at 16 days with 10 meter resolution) and "MOD13Q1-6" (MODIS MOD13SQ1 product, collection 6).

```
cbers_tile <- sits_cube(
    source = "BDC",
    collection = "CB4_64_16D_STK-1",
    name = "cbers_022024",
    bands = c("NDVI", "EVI"),
    tiles = "022024",
    start_date = "2018-09-01",
    end_date = "2019-08-28"
)
```

1.3.3 Defining a data cube using files

In some cases, users have downloaded files from image collections and have them available in their computer or in a local network. As *sits* does not have access to STAC information that describe the files, they should be organized and named to allow SITS to create a data cube.

All 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. Since raster files in popular formats (e.g., GeoTiff and JPEG 2000) do not include temporal and band information, each file name needs to include date and band. Information on the tile reference system should be provided. Also, When building data cubes from images stored in a local machine, users need to provide information about the original source from with the data was downloaded. The reason to included such information is because there are no standards for the metadata used to process an image.

When working with local cubes, the following parameters should be provided to

the `sits_cube()` function:

- `source` - value should be "LOCAL";
- `name` - internal name for the data cube (free user choice);
- `origin` - name of the original data (possible values are "BDC" (Brazil Data Cube), "AWS" (Amazon Web Services), or "DEA" (Digital Earth Africa));
- `collection` - name of the collection from where the data was extracted. This should be "MOD13Q1-6," "CB4_64_16D_STK-1," "LC8_30_16D_STK-1" and "S2-SEN2COR_10_16D_STK-1," respectively, for CBERS-4, LANDSAT-8, and SENTINEL-2 in the case of the BDC. For "AWS," the only collection currently supported is "SENTINEL-S2-L2A." In the case of "DEA," the supported collection is "S2_L2A." More collections will be added in future versions of the ;
- `data_dir` - directory where the image data is located;
- `parse_info` - information to parse the file names and extract the information on the tile, date and band associated with each individual file. It is assumed that file names contain image descriptors separated by a delimiter (usually "_"). For example, `CBERS-4_AWFI_022024_B13_2018-02-02.tif` and `SENTINEL-2_MSI_L20KP_20m_B08_2021_03_29.jp2` are valid file names for sits. The `parse_info` parameter is a list of strings indicating at which position the names of `tile`, `date` and `band` are to be found. In the two file names above, the parsing info is respectively `c("X1", "X2", "tile", band", "date")` and `c("X1", "X2", "tile", "X3", "band", "date")`;
- `delim` - separator character between descriptions in the file name (default is "_").

The following example shows how to define a data cube based on local files available as part of the *sits* package.

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

# files are named using the convention
# "CB4_64_16D_STK_022024_2018-08-29_2018-09-13_EVI.tif"
cbers_cube <- sits_cube(
      source = "LOCAL",
      name = "022024",
      origin = "BDC",
      collection = "MOD13Q1-6",
      data_dir = data_dir,
      delim = "_",
      parse_info = c("X1", "X2", "X3", "X4", "tile", "date", "X5", "band")
)
# show the timeline of the cube
```

```
sits_timeline(cbers_cube)
# show the bands of the cube
sits_bands(cbers_cube)
# plot the band B16 in the first time instance
plot(cbers_cube, band = "NDVI", time = 1)
```

1.4 Regularizing data cubes

Analysis-ready data (ARD) collections available in AWS and DE Africa do not have consistent timelines. In general, images in neighboring tiles have different timelines. This is a problem when classifying large areas. In this case, users may want to produce data cubes with regular time intervals. For example, a user may want to define the best Sentinel-2 pixel in a one-month period. This can be done in *sits* by the function `sits_regularize()`, which uses the package *gdalcubes* [3]. For details in gdalcubes, please see https://github.com/appelmar/gdalcubes.

```
# creating a data cube in AWS
s2_cube <- sits_cube(source = "AWS",
                     name = "T20LKP_2018_2019",
                     collection = "sentinel-s2-l2a",
                     tiles = c("20LKP", "20LLP"),
                     start_date = as.Date("2018-07-01"),
                     end_date = as.Date("2018-08-31"),
                     bands = c("B11", "CLOUD"),
                     s2_resolution = 60
)
# list the timeline of the AWS S2 cube (there are 12 images)
sits_timeline(s2_cube)
# regularize the cube to one month intervals
reg_cube <- sits_regularize(
          cube       = s2_cube,
          name       = "T20LKP_20LKP_20LLP_15D",
          output_dir = tempdir(),
          res = 60,
          period     = "P15D",
          agg_method = "median",
          cloud_mask = TRUE,
          multicores = 4

)
```

In the above example, the user has selected the `s2_cube` object defined using AWS (see example above). As described earlier in this chapter, because of the way ARD image collections are built, the timelines of tiles "20LLP" and "20LKP" associated with this cube are different. The `sits_regularize()` function builds a new data

cube, with the same temporal extent as the `s2_cube` but with the same timeline. In this function, the `period` parameter controls the temporal interval between two images. Values should abide by the ISO8601 time period specification, which states that time interval should be defined 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, defined by the parameter `agg_method`. It specifies how individual values of different pixels should be combined. The default is `median`, which select the most frequent value for the pixel during the desired interval. For more details, see `?sits_regularize`.

Chapter 2

Working with time series

This chapter describes how to access information from time series in SITS.

2.1 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 `sits 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]. As an example of how the sits tibble works, 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," "Soybean-fallow," "Fallow-Cotton," "Soybean-Cotton," "Soybean-Corn," "Soybean-Millet," and "Soybean-Sunflower."

```
# data set of samples
library(sits)
```

```
data("samples_matogrosso_mod13q1")
samples_matogrosso_mod13q1[1:3,]
```

```
#> # A tibble: 3 x 7
#>   longitude latitude start_date end_date   label   cube    time_series
#>       <dbl>    <dbl> <date>     <date>     <chr>   <chr>   <list>
#> 1     -55.2   -10.8  2013-09-14 2014-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 2     -57.8    -9.76 2006-09-14 2007-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 3     -51.9   -13.4  2014-09-14 2015-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
```

A sits tibble 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 spatial location is given in longitude and latitude coordinates for the "WGS84" ellipsoid. For example, the first sample has been labeled "Cerrado, at location (-58.5631, -13.8844) and is considered valid for the period (2007-09-14, 2008-08-28). 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, where the spring crop is planted in the months of September and October, and the autumn crop is planted in the months of February and March. 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.

2.2 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 either as "Cerrado" or "Pasture."

```
# 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")
```

2.3 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,])
```

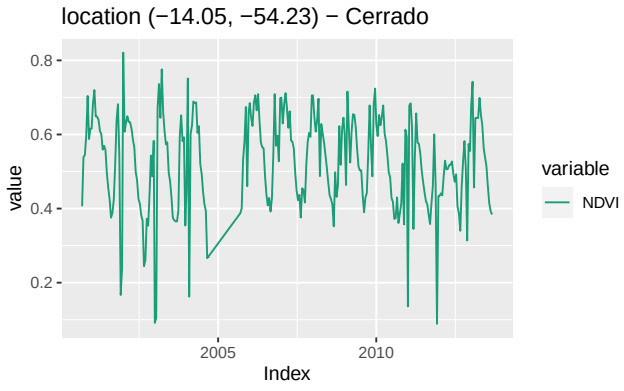


Figure 2.1: Plot of the first 'Cerrado' sample

For a large number of samples, where the number of individual plots would be

substantial, the default visualization combines all samples together in a single temporal interval (even if they belong to different years). All samples with the same band and label are aligned to a common time interval. 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. The documentation of `plot.sits()` has more details about the different ways it can display data.

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

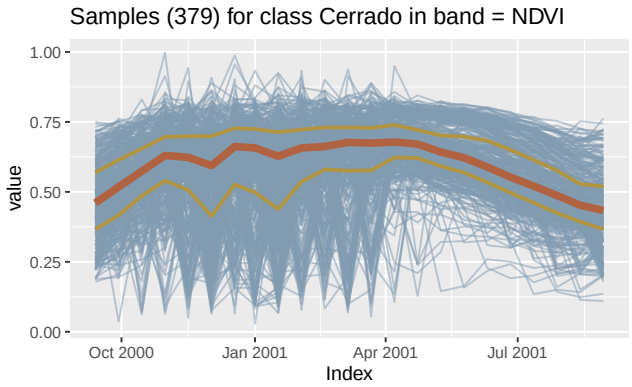


Figure 2.2: Plot of all Cerrado samples

2.4 Obtaining time series data from data cubes

To get a time series in sits, one has to create a data cube, as described previously. Users can request one or more time series points from a data cube by using `sits_get_data()`. This function provides a general means of access to image time series. Given a data cube, the user provides the latitude and longitude of the desired location, the bands, and the start date and end date of the time series. If the start and end dates are not provided, it retrieves all the available periods. The result is a tibble that can be visualized using `plot()`.

```
# Obtain a raster cube with 23 instances for one year
# Select the band "ndvi", "evi" from images available in the "sitsdata" package
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     = "LOCAL",
    origin     = "BDC",
    collection = "MOD13Q1-6",
```

```
    name       = "Sinop",
    data_dir   = data_dir,
    parse_info = c("X1", "X2", "tile", "band", "date")
)
# a point in the transition forest to pasture in Northern MT
# obtain a time series from the raster cube for this point
series <- sits_get_data(cube      = raster_cube,
                        longitude = -55.57320,
                        latitude  = -11.50566,
                        bands     = c("NDVI", "EVI"))
plot(series)
```

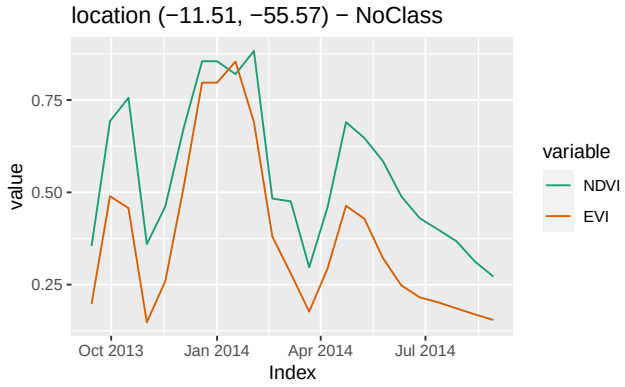


Figure 2.3: NDVI and EVI time series fetched from local raster cube.

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). Function `sits_get_data()` takes a CSV or SHP file path as an argument. For each training sample, CSV files should provide latitude and longitude, start and end dates that define the temporal bounds, and a label associated with a ground sample. An example of a CSV file used is shown below.

```
# 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,]
```

```
#>   id longitude  latitude start_date   end_date   label
#> 1  1 -55.65931 -11.76267 2013-09-14 2014-08-29 Pasture
```

```
#> 2  2 -55.64833 -11.76385 2013-09-14 2014-08-29 Pasture
#> 3  3 -55.66738 -11.78032 2013-09-14 2014-08-29  Forest
```

The main difference between the files used by *sits* to retrieve training samples from those used traditionally in remote sensing data analysis is that users are expected 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_modis_4bands` provided with the sits 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_modis_4bands` have the same number of instances.

```
samples_modis_4bands[1:5,]
```

```
#> # A tibble: 5 x 7
#>   longitude latitude start_date end_date   label   cube    time_series
#>       <dbl>    <dbl> <date>     <date>     <chr>   <chr>   <list>
#> 1     -55.2   -10.8  2013-09-14 2014-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 2     -57.8    -9.76 2006-09-14 2007-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 3     -51.9   -13.4  2014-09-14 2015-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 4     -56.0   -10.1  2005-09-14 2006-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
#> 5     -54.6   -10.4  2013-09-14 2014-08-29 Pasture MOD13Q1 <tibble [23 x 5]>
```

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) `file`, the name of the CSV file.

```
# get the points from a data cube in raster brick format
points <- sits_get_data(cube = raster_cube, file = 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  time_series
#>       <dbl>    <dbl> <date>     <date>     <chr>   <chr> <list>
#> 1     -55.7    -11.8 2013-09-14 2014-08-29 Pasture Sinop <tibble [23 x 3]>
#> 2     -55.6    -11.8 2013-09-14 2014-08-29 Pasture Sinop <tibble [23 x 3]>
#> 3     -55.7    -11.8 2013-09-14 2014-08-29 Forest  Sinop <tibble [23 x 3]>
```

Users can also specify samples by providing shapefiles in point or polygon format. In this case, the geographical location is inferred from the geometries associated with the shapefile. For files containing points, the geographical location is obtained directly; for files with polygon, the parameter `.n_shp_pol` (defaults to 20) determines the number of samples to be extracted from each polygon. The temporal information is inferred from the data cube from which the samples are extracted or can be provided explicitly by the user. The label information is taken from the attribute file associated with the shapefile. The parameter

`shp_attr` indicates the name of the column which contains the label to be associated with each time series.

```
# define the input shapefile
shp_file <- system.file("extdata/shapefiles/agriculture/parcel_agriculture.shp",
                        package = "sits")

# set the start and end dates
start_date <- "2013-09-14"
end_date   <- "2014-08-29"

# define the attribute name that contains the label
shp_attr <- "ext_na"

# define the number of samples to extract from each polygon
.n_shp_pol <- 10

# read the points in the shapefile and produce a CSV file
samples <- sits_get_data(cube       = raster_cube,
                         file       = shp_file,
                         start_date = start_date,
                         end_date   = end_date,
                         shp_attr   = shp_attr,
                         .n_shp_pol = .n_shp_pol)
samples[1:3,]
```

```
#> # A tibble: 3 x 0
```

2.5 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.

2.5.1 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 central point i of the window array assumes the value of the interpolated polynomial. An equivalent and much faster solution than this convolution procedure is given by the closed expression

$$\hat{x}_i = \sum_{j=-n}^{n} C_j \, x_{i+j},$$

where $\hat{x}$ is the the filtered time series, C_j are the Savitzky-Golay smoothing coefficients, and x is the original time series.

The coefficients C_j depend uniquely on the polynomial degree (d) and the length of the window data points (given by parameter n). If $d = 0$, the coefficients are constants $C_j = 1/(2n + 1)$ and the Savitzky-Golay filter will be equivalent to moving average filter. When the time series are equally spaced, the coefficients have an analytical solution. According to [8], for $d \in [2, 3]$ each C_j smoothing coefficients can be obtained by

$$C_j = \frac{3(3n^2 + 3n - 1 - 5j^2)}{(2n + 3)(2n + 1)(2n - 1)}.$$

In general, the Savitzky-Golay filter produces smoother results for a larger value of n and/or a smaller value of d [9]. 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 = 15)
# 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 remove noise resulting from 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

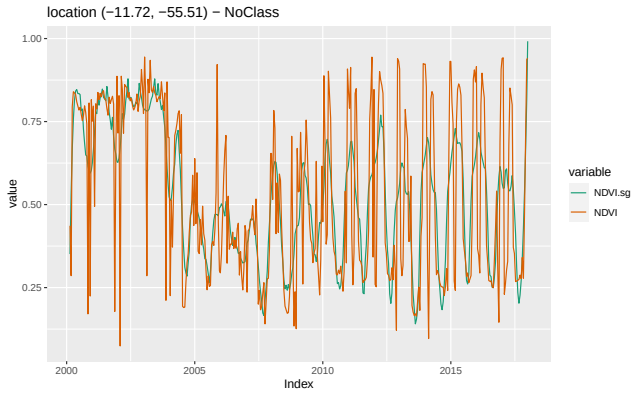


Figure 2.4: Savitzky-Golay filter applied on a multi-year NDVI time series.

results in oversmoothing. Users can try to reduce this parameter and analyse the results.

2.5.2 Whittaker filter

The Whittaker smoother attempts to fit a curve that represents the raw data, but is penalized if subsequent points vary too much [10]. The Whittaker filter is a balancing between the residual to the original data and the "smoothness" of the fitted curve. The residual, as measured by the sum of squares of all n time series points deviations, is given by

$$RSS = \sum_i (x_i - \hat{x_i})^2,$$

where x and $\hat{x}$ are the original and the filtered time series vectors, respectively. The smoothness is assumed to be the measure of the sum of the squares of the third-order differences of the time series [11], which is given by

$$\begin{aligned} SSD = (\hat{x}_4 - 3\hat{x}_3 + 3\hat{x}_2 - \hat{x}_1)^2 + (\hat{x}_5 - 3\hat{x}_4 + 3\hat{x}_3 - \hat{x}_2)^2 \\ + \ldots + (\hat{x}_n - 3\hat{x}_{n-1} + 3\hat{x}_{n-2} - \hat{x}_{n-3})^2. \end{aligned}$$

The filter is obtained by finding a new time series $\hat{x}$ whose points minimize the expression

$$RSS + \lambda SSD,$$

where λ, a scalar, works as a "smoothing weight" parameter. The minimization can be obtained by differentiating the expression with respect to $\hat{x}$ and equating it to zero. The solution of the resulting linear system of equations gives the filtered time series, which, in matrix form, can be expressed as

$$\hat{x} = (\mathrm{I} + \lambda D^{\intercal} D)^{-1} x,$$

where I is the identity matrix and

$$D = \begin{bmatrix} 1 & -3 & 3 & -1 & 0 & 0 & \cdots \\ 0 & 1 & -3 & 3 & -1 & 0 & \cdots \\ 0 & 0 & 1 & -3 & 3 & -1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

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()
```

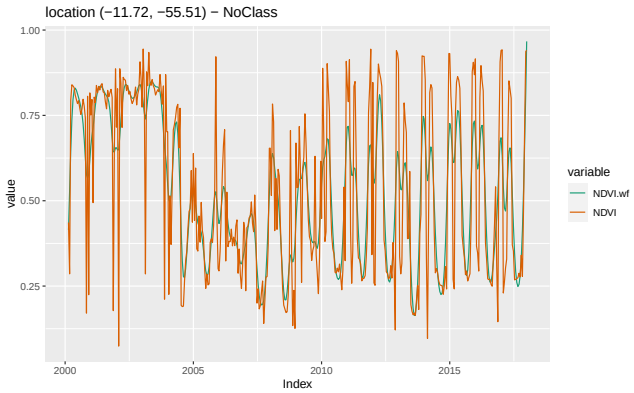


Figure 2.5: Whittaker filter applied on a one-year NDVI time series.

In the same way as what is observed in the Savitsky-Golay filter, high values of the smoothing parameter `lambda` produce an oversmoothed 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.

Part I

Clustering

Chapter 3

Time Series Clustering to Improve the Quality of Training Samples

---

One of the key challenges when using samples to train machine learning classification models is assessing their quality. Noisy and imperfect training samples can have a negative effect on classification performance. 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. `sits` provides two clustering methods to improve sample quality: agglomerative hierarchical clustering (AHC) and self-organizing maps (SOM).

3.1 Clustering for sample quality control: overview

Experience with machine learning methods has established 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 [12]; noisy training samples can have a negative effect on classification performance [13].

When assessing the quality of training samples, it is useful to distinguish between samples that have been wrongly labelled 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 can result in different patterns being assigned to the same label. Phenological patterns can

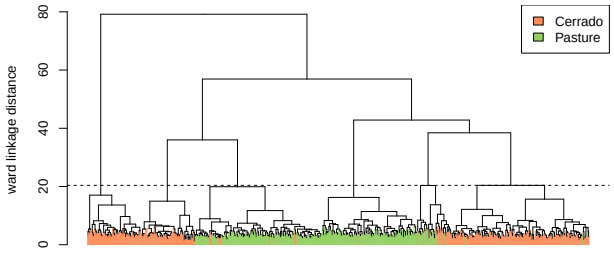
vary spatially across a region and are strongly correlated with climate variations. A related issue is the limitation of crisp boundaries to describe the natural world. Class definition use idealized descriptions (e.g., "a savanna woodland has tree cover of 50% to 90% ranging from 8 to 15 meters in height"). In practice, the boundaries between classes are fuzzy and sometimes overlap, making it hard to distinguish between them. 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. Representative samples lead to good classification maps. 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. Therefore, for large data sets, AHC requires an substantial amount of memory and running time; in these cases, SOM is recommended.

3.2 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 [14]. 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"
)
```



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)[15], as DTW is reliable method for measuring differences between satellite image time series [16]. 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. Complete-linkage clustering can be sensitive to outliers, which can increase the resulting intracluster data variance. As an alternative, Ward proposes criteria to minimize the data variance by means of either *sum-of-squares* or *sum-of-squares-error* [17]. Ward's intuition is that clusters of multivariate observations, such as time series, should be approximately elliptical in shape [18].

After creating a dendrogram, an important question emerges: *where to cut the dendrogram?* The answer depends on what are the purposes of the cluster analysis, which needs to balance two objectives: get clusters as large as possible, and get clusters as homogeneous as possible with respect to their known classes. The `sits_cluster_dendro()` function computes the *adjusted rand index* (ARI) for a series of the different number of generated clusters. This function returns the height where the cut of the dendrogram maximizes the index. For more detaily, please see [19].

In the example above after calculating the dendrogram, the ARI index indicates that six (6) clusters are the best possible arrangement. However, these clusters may still contain a mixed composition of samples of different classes.

The result of the `sits_cluster` operation is a `sits_tibble` with one additional

column, called "cluster." The function `sits_cluster_frequency()` provides information on the composition of each cluster,

```
clusters
```

```
#> # A tibble: 746 x 8
#>    longitude latitude start_date end_date   label   cube    time_series       cluste
#>        <dbl>    <dbl> <date>     <date>     <chr>   <chr>   <list>              <int
#>  1     -54.2    -14.0 2000-09-13 2001-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  2     -54.2    -14.0 2001-09-14 2002-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  3     -54.2    -14.0 2002-09-14 2003-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  4     -54.2    -14.0 2003-09-14 2004-08-28 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  5     -54.2    -14.0 2004-09-13 2005-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  6     -54.2    -14.0 2005-09-14 2006-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  7     -54.2    -14.0 2006-09-14 2007-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  8     -54.2    -14.0 2007-09-14 2008-08-28 Cerrado MOD13Q1 <tibble [23 x 3]>
#>  9     -54.2    -14.0 2008-09-13 2009-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#> 10     -54.2    -14.0 2009-09-14 2010-08-29 Cerrado MOD13Q1 <tibble [23 x 3]>
#> # ... with 736 more rows
```

```
# 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 result shows that the clusters have a predominance of either "Cerrado" or "Pasture" classes with the exception of cluster 3. The contingency table plotted by `sits_cluster_frequency()` shows how the samples are distributed across the clusters and helps identify two kinds of problems. The first is relative to small amounts of samples in clusters dominated by another class (*e.g.* clusters 1, 2, 4, 5, and 6), while the second is relative to those samples in non-dominated clusters (*e.g.* cluster 3). These confusions can be an indication of samples with poor quality, and inadequacy of selected parameters for cluster analysis, or even a natural confusion due to the inherent variability of the land classes.

It is possible to remove clusters with mixed classes using the `dplyr` package. In the example above, removing cluster 3 can be done using the `dplyr::filter()` function.

```
# remove cluster 3 from the samples
clusters_new <- dplyr::filter(clusters, cluster != 3)

# show new clusters samples frequency
sits::sits_cluster_frequency(clusters_new)
```

```
#>
#>              1   2   4   5   6 Total
#>   Cerrado 203  13  80   1  80   377
#>   Pasture   2 176   0 140   0   318
#>   Total   205 189  80 141  80   695
```

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. To do this, one can use `sits_cluster_clean()`, which removes all minority samples, as shown below.

```
# clear clusters, leaving only the majority class in each cluster
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
```

After cleaning the samples using dendrograms, users are expected to have a better set of samples which will provide more accurate estimates of land classification.

3.3 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 [20], 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 (BMU).

The input data for quality assessment is a set of training samples. Training samples associated to satellite image time series are high-dimensional data sets. A time series of 25 instances of 4 spectral bands has 100 dimensions. When projecting a high-dimensional data set of training samples 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

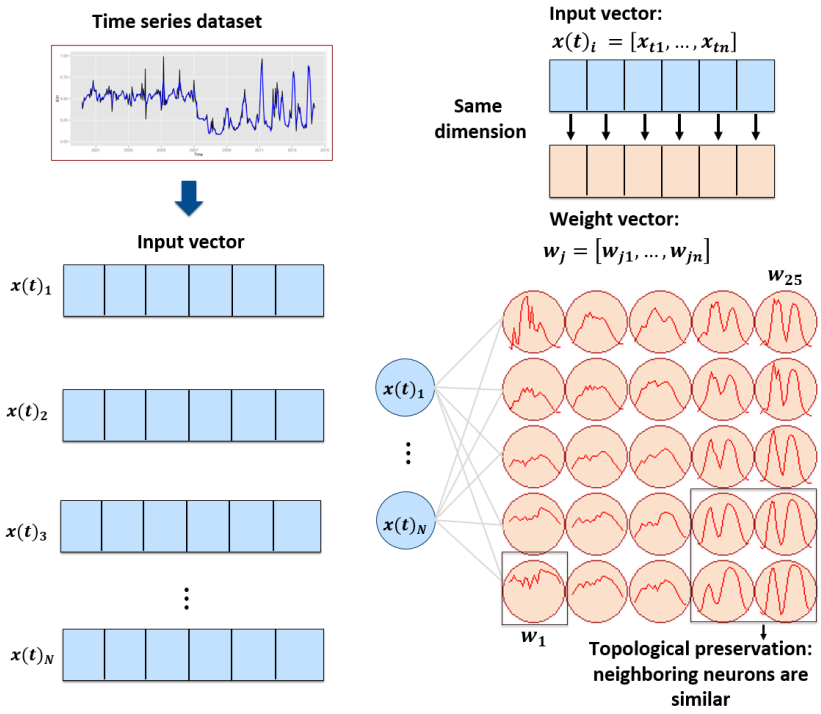


Figure 3.1: SOM 2D map creation (source: Santos et al.(2021)

topological structure of neighborhoods in multiple dimensions, clusters that contain training samples of a given class will usually be neighbors in 2D space. Therefore, the neighbours 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 [21].

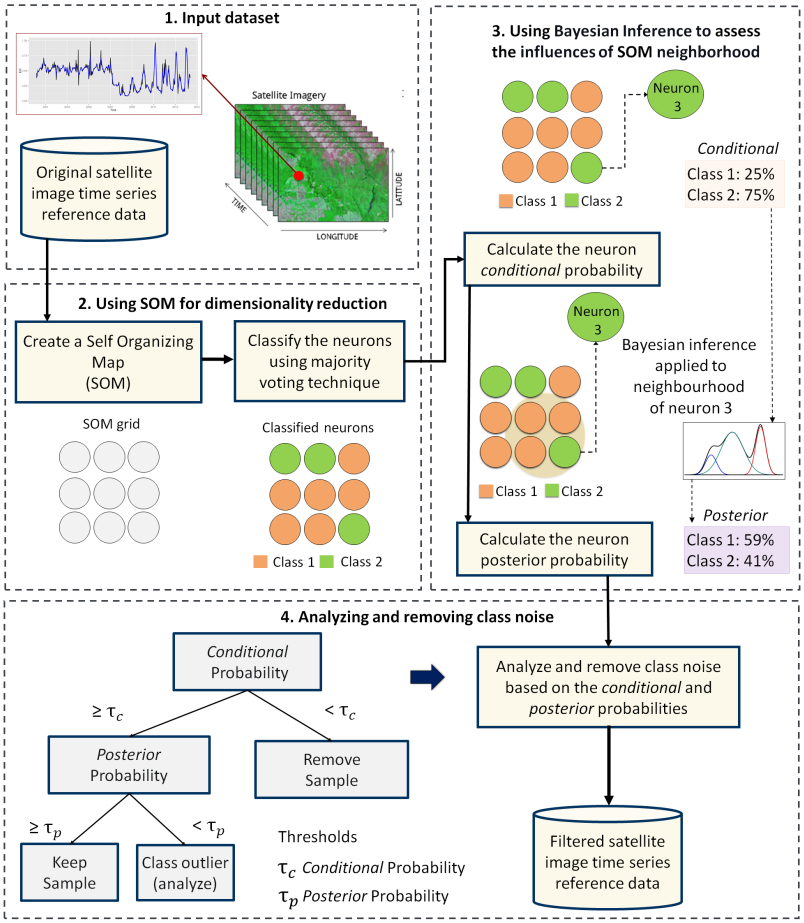


Figure 3.2: Using SOM for class noise reduction (source: Santos et al.(2021)

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. The training samples were collected by ground surveys and high-resolution image interpretation by experts from the Brazilian National Institute for Space Research (INPE) team and partners. 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").

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

3.3.1 SOM-based quality assessment part 1: creating the SOM map

To run the SOM-based quality assessment, the first step is to run `sits_som_map()` which uses the "kohonen" R package [22] to compute a SOM grid. Each sample is assigned to a neuron, and neurons are placed in the grid based on similarity. This function has six main parameters. In `data`, the user should provide the name of the R object containing the samples. The size of the SOM map grid is controlled by `grid_xdim` and `grid_ydim`. The starting learning rate is set using `alpha`; this learning rate decreases during the interactions. The distance metric is controlled by `distance`; options available currently are "sumofsquares" and "euclidean." The number of iterations is set by `rlen`. For more details on the implementation, please also consult `?kohonen::supersom`.

```
# clustering time series using SOM
som_cluster <-
    sits_som_map(
```

```
        data = samples_cerrado_mod13q1_2bands,
        grid_xdim = 15,
        grid_ydim = 15,
        alpha = 1.0,
        distance = "euclidean",
        rlen = 20
    )
```

The output of the `sits_som_map()` is a list with 4 tibbles:

- 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).
- a tibble 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.
- the SOM grid

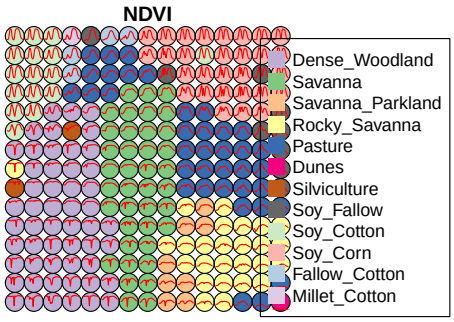
```
som_cluster
```

```
#> $data
#> # A tibble: 50,160 x 9
#>    latitude longitude start_date end_date   label   cube    time_series id_sample id_neuron
#>       <dbl>     <dbl> <date>     <date>     <chr>   <chr>   <list>          <int>     <dbl>
#>  1    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         1       115
#>  2    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         2       115
#>  3    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         3       115
#>  4    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         4       131
#>  5    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         5       188
#>  6    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         6       145
#>  7    -16.2     -54.4 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         7       116
#>  8    -16.2     -54.5 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         8       130
#>  9    -16.1     -54.7 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~         9       130
#> 10    -16.2     -54.7 2018-09-01 2019-09-01 Pasture MOD13Q1 <tibble [2~        10       188
#> # ... with 50,150 more rows
#>
#> $labelled_neurons
#> # A tibble: 746 x 5
#>    id_neuron label_samples  count prior_prob post_prob
#>        <dbl> <chr>          <int>      <dbl>     <dbl>
#>  1         1 Dense_Woodland   447    0.982     0.905
#>  2         1 Pasture            1    0.00220   0.0426
#>  3         1 Rocky_Savanna      1    0.00220   0.0462
#>  4         1 Savanna            5    0.0110    0.00170
#>  5         1 Silviculture       1    0.00220   0.0223
#>  6         2 Dense_Woodland   203    0.953     0.846
#>  7         2 Pasture            8    0.0376    0.0746
```

```
#>  8         2 Savanna            2    0.00939   0.00170
#>  9         3 Dense_Woodland   139    0.586     0.842
#> 10         3 Pasture           95    0.401     0.0493
#> # ... with 736 more rows
#>
#> $som_properties
#> SOM of size 15x15 with a rectangular topology.
#> Training data included.
#>
#> attr(,"class")
#> [1] "som_map" "list"
```

To plot the SOM grid, use `plot()`. The neurons are labelled using the majority voting.

```
plot(som_cluster)
```



The SOM grid has 225 neurons, as defined by the product of the parameters `grid_xdim` and `grid_ydim` in the `sits_som_map()`. Looking at the SOM grid, one can see that most of the neurons of a class are located close to each other, as expected. However, some "Pasture" neurons far from the main cluster. This mixture is a consequence of the continuous nature of natural vegetation cover in the Brazilian Cerrado. The transition between areas of open savanna and pasture is not always well defined; moreover, it is dependent on factors such as climate and latitude. The SOM grid provides a general view of the capacity of the samples to distinguish the chosen labels.

3.3.2 SOM-based quality assessment part 2: assessing 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)
```

```
#> Joining, by = "id_neuron"
```

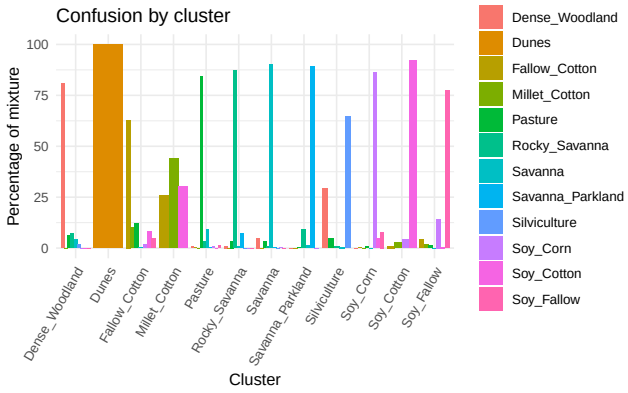
```
# show the result
som_eval
```

```
#> # A tibble: 79 x 4
#>    id_cluster cluster        class          mixture_percentage
#>         <int> <chr>          <chr>                       <dbl>
#>  1          1 Dense_Woodland Dense_Woodland           81.0
#>  2          1 Dense_Woodland Millet_Cotton             0.00871
#>  3          1 Dense_Woodland Pasture                   6.14
#>  4          1 Dense_Woodland Rocky_Savanna             7.16
#>  5          1 Dense_Woodland Savanna                   4.03
#>  6          1 Dense_Woodland Silviculture              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 ("Soy_Corn," "Soy_Cotton" and "Soy_Fallow").

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



3.3.3 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. The probability information is included in the `labelled_neurons` tibble which is produced by `sits_som_map()` (see above). 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 each 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 are not 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? Are the samples labeled "Pasture" in this neuron noisy? To answer this question, we use information from the neighbors. Using Bayesian inference, we estimate if these samples are noisy based on the samples of the neighboring neurons [23].

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 `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 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 of being noisy and the assigned label. The default value 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_cluster,
                                      prior_threshold = 0.6,
                                      posterior_threshold = 0.6,
                                      keep = c("clean", "analyze"))
# find out how many samples are evaluated as "clean" or "analyze"
new_samples %>%
  dplyr::group_by(eval) %>%
  dplyr::summarise(count = dplyr::n(), .groups = "drop")
```

```
#> # A tibble: 2 x 2
#>   eval    count
#>   <chr>   <int>
#> 1 analyze  5971
#> 2 clean   33691
```

3.3.4 Comparing Original and Clean Samples

To compare the original and clean samples, we run a 5-fold validation on the original and on the cleaned sample set, using the function `sits_kfold_validate()` and a random forest model. As the results show, the SOM procedure is useful, since the validation improves from 95% to 99%. It should be noted that a k-fold validation procedure measures how well the machine learning model fits the samples and is not a measure of accuracy of the classification. For more details on accuracy measures, please see Chr 8.

```
# run a k-fold validation
assess_orig <- sits_kfold_validate(samples_cerrado_mod13q1_2bands,
                                   ml_method = sits_rfor(num_trees = 200))
# print summary
```

```
sits_accuracy_summary(assess_orig)
```

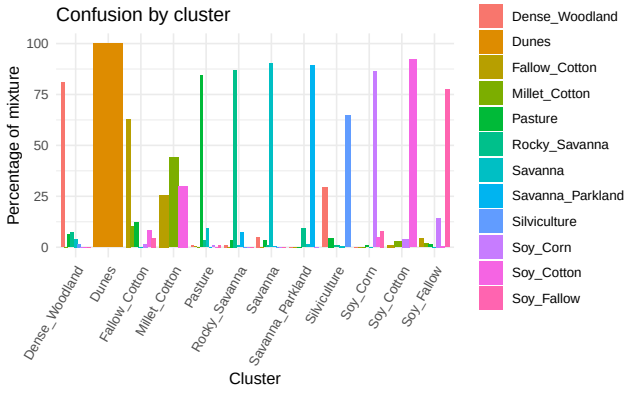
```
#>
#> Overall Statistics
#>
#>  Accuracy : 0.9467
#>    95% CI : (0.9447, 0.9486)
#>
#>     Kappa : 0.9378
```

```
assess_new <- sits_kfold_validate(new_samples,
                                  ml_method = sits_rfor(num_trees = 200))
# print summary
sits_accuracy_summary(assess_new)
```

```
#>
#> Overall Statistics
#>
#>  Accuracy : 0.9914
#>    95% CI : (0.9904, 0.9923)
#>
#>     Kappa : 0.9899
```

An additional way of evaluating the quality of samples is to examine the internal mixture inside neurons with the same label. We call a group of neurons sharing the same label as a "cluster." Given a SOM map, the function `sits_som_evaluate_cluster` examines all clusters to find out the percentage of samples contained in it which do not share its label. This information is saved as a tibble and can also be visualized.

```
# evaluate the misture in the SOM clusters
cluster_mixture <- sits_som_evaluate_cluster(som_cluster)
# plot the mixture information.
plot(cluster_mixture)
```



3.4 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, this paper proposed a new technique. The proposed method uses the SOM neural network to group similar samples in a 2D map for dimensionality reduction. The method identifies both mislabeled samples and outliers that are flagged to further investigation. The results demonstrate the positive impact on the overall classification accuracy. Although the class noise removal adds an extra cost to the entire classification process, we believe that it is essential to improve the accuracy of classified maps using SITS analysis mainly for large areas.

Part II

Classification

Chapter 4

Machine Learning for Data Cubes using the SITS package

This chapter presents the machine learning (ML) techniques available in SITS. The main use for machine learning in SITS is for classification of land use and land cover. These machine learning methods available in SITS include linear and quadratic discrimination analysis, support vector machines, random forests, deep learning and neural networks.

4.1 Machine learning classification

The package provides support for the classification of time series, preserving the full temporal resolution of the input data. Instead of extracting metrics from time series segments, it uses all values of the time series. It supports two kinds of machine learning methods. The first group of methods does not explicitly consider spatial or temporal dimensions; these models treat time series as a vector in a high-dimensional feature space. From this class of models, `sits` includes random forests[24], support vector machines[25], extreme gradient boosting[26], and multi-layer perceptrons[27].

The second group of models comprises deep learning methods specifically designed to work with time series. Temporal relations between observed values in a time series are taken into account. Time series classification models for satellite data include 1D convolution neural networks (1D-CNN) [28], recurrent neural

networks (RNN)[29], and~attention-based deep learning [31]. The `sits` package supports two 1D-CNN algorithms: TempCNN[2] and ResNet[32]. For models based on 1D-CNN, the order of the samples in the time series is relevant for the classifier. Each layer of the network applies a convolution filter to the output of the previous layer. This cascade of convolutions captures time series features in different time scales [2].

Thus, the following machine learning methods are available in SITS:

- Support vector machines (`sits_svm()`)
- Random forests (`sits_rfor()`)
- Extreme gradient boosting (`sits_xgboost()`)
- Deep learning (DL) using multi-layer perceptrons (`sits_mlp()`)
- DL using Deep Residual Networks (`sits_ResNet()`)
- DL with 1D convolutional neural networks (`sits_TempCNN()`)

For the machine learning examples, we use the data set "samples_matogrosso_mod13q1," containing a sits tibble with time series samples from Brazilian Mato Grosso State (Amazon and Cerrado biomes), obtained from the MODIS MOD13Q1 product. The tibble with 1,892 samples and 9 classes ("Cerrado," "Fallow_Cotton," "Forest," "Millet_Cotton," "Pasture,""Soy_Corn," "Soy_Cotton," "Soy_Fallow," "Soy_Millet"). Each time series comprehends 12 months (23 data points) with 6 bands ("NDVI," "EVI," "BLUE," "RED," "NIR," "MIR"). The dataset was used in the paper "Big Earth observation time series analysis for monitoring Brazilian agriculture" [33], and is available in the R package "sitsdata," which is downloadable from the website associated to the "e-sensing" project.

The following examples show how to train ML and apply it to classify a single time series; they 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 [12]. Given a good set of samples, users of `sits` should be able to get good results. For examples of ML for classifying large areas, please see Chapter 6 and the papers by the authors [36].

4.2 Visualizing Samples

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 Chapter 3. In the plot, the thick red line is the median value for each time instance and the yellow lines are the first and third interquartile ranges.

```
data("samples_matogrosso_mod13q1")
# Select a subset of the samples to be plotted
# Retrieve the set of samples for the Mato Grosso region
samples_matogrosso_mod13q1 %>%
    sits_select(bands = "NDVI") %>%
    dplyr::filter(label == "Forest") %>%
```

```
    plot()
```

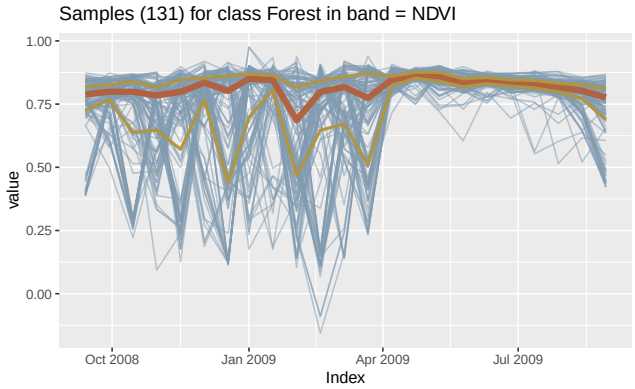


Figure 4.1: Visualisation of samples associated to class Forest in band NDVI

An alternative to visualise the samples is to estimate a statistical approximation to an idealized pattern based on a generalised additive model (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`[37], which implements the TWDTW time series matching method described in [16]. The resulting patterns can be viewed using `plot`.

```
# Select a subset of the samples to be plotted
samples_matogrosso_mod13q1 %>%
    sits_patterns() %>%
    plot()
```

The resulting plots provide some insights over the time series behaviour of each class. While the response of the "Forest" class is quite distinctive, there are similarities between the double-cropping classes ("Soy-Corn," "Soy-Millet," "Soy-Sunflower" and "Soy-Corn") and between the "Cerrado" and "Pasture" classes. This could suggest that additional information, more bands, or higher-resolution data could be considered to provide a better basis for time series samples that can better distinguish the intended classes. Despite these limitations, the best machine learning algorithms can provide good performance even in the above case.

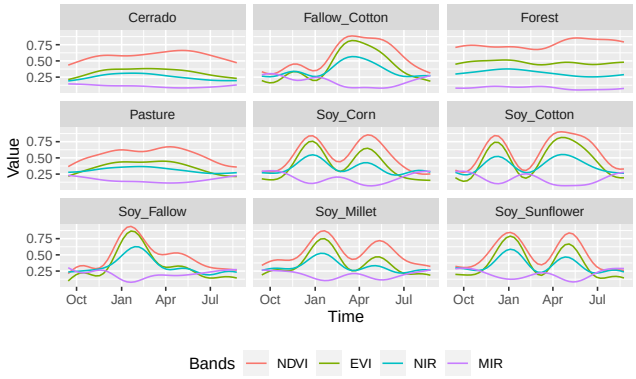


Figure 4.2: Patterns for the samples for Mato Grosso

4.3 Common interface to machine learning and deeplearning 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 input data samples and the ML method (`ml_method`), as shown below. After the model is estimated, it can be used to classify individual time series or full data cubes using `sits_classify()`. In the examples that follow, we show how to apply each method for the classification of a single time series. Then, in Chapter 6 we disscuss how to classify data cubes.

Since `sits` is aimed at remote sensing users who are not machine learning experts, the~package provides a set of default values for all classification models. These settings have been chosen based on extensive testing by the authors. Nevertheless, users can control all parameters for each model. The package documentation describes in detail the tuning parameters for all models that are available in the respective functions. Thus, novice users can rely on the default values, while experienced ones can fine-tune model parameters to meet their needs.

When a dataset of time series organised as a SITS 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()`.

4.4 Random forests

The random forest model uses the idea of *decision trees* as its base model, with many 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 sample set[38]. Each of these features is then tested, the one maximizing the decrease in a purity measure 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 the prediction performance. The classification performance depends on the number of trees in the forest as well as the number of features randomly selected at each node. SITS provides a `sits_rfor` function which is a front-end to the `randomForest` package[39]; its main parameter is `num_trees` (number of trees to grow). In practice, between 100 and 200 trees are sufficient to achieve a reasonable classification result.

```
# Retrieve the set of samples (provided by EMBRAPA) from the
# Mato Grosso region for train the Random Forest model.
rfor_model <- sits_train(data = samples_matogrosso_mod13q1,
                         ml_method = sits_rfor(num_trees = 200))
# retrieve a point to be classified
point_mt_4bands <- sits_select(point_mt_6bands,
                               bands = c("NDVI", "EVI", "NIR", "MIR"))
# Classify using Random Forest model and plot the result
point_class <- sits_classify(point_mt_4bands, rfor_model)
plot(point_class, bands = c("NDVI", "EVI"))
```

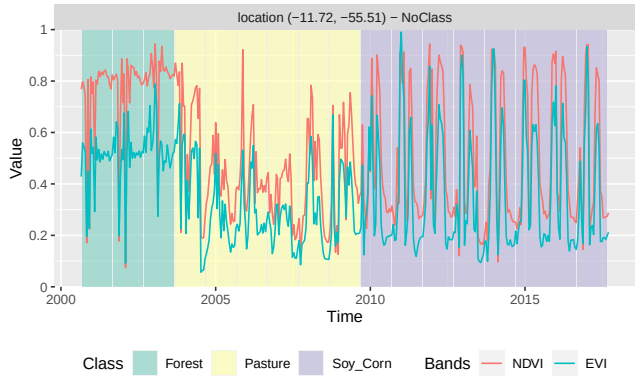


Figure 4.3: Classification of time series using random forests

```
# show the results of the prediction
sits_show_prediction(point_class)
```

```
#> # A tibble: 17 x 3
#>    from       to         class
#>    <date>     <date>     <chr>
#>  1 2000-09-13 2001-08-29 Forest
#>  2 2001-09-14 2002-08-29 Forest
```

```
#>  3 2002-09-14 2003-08-29 Forest
#>  4 2003-09-14 2004-08-28 Pasture
#>  5 2004-09-13 2005-08-29 Pasture
#>  6 2005-09-14 2006-08-29 Pasture
#>  7 2006-09-14 2007-08-29 Pasture
#>  8 2007-09-14 2008-08-28 Pasture
#>  9 2008-09-13 2009-08-29 Pasture
#> 10 2009-09-14 2010-08-29 Soy_Corn
#> 11 2010-09-14 2011-08-29 Soy_Corn
#> 12 2011-09-14 2012-08-28 Soy_Corn
#> 13 2012-09-13 2013-08-29 Soy_Corn
#> 14 2013-09-14 2014-08-29 Soy_Corn
#> 15 2014-09-14 2015-08-29 Soy_Corn
#> 16 2015-09-14 2016-08-28 Soy_Corn
#> 17 2016-09-13 2017-08-29 Soy_Corn
```

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 [40]. However, despite being robust, this approach tends to overemphasize some variables and thus rarely turns out to be the classifier with the smallest error. One reason is that the performance of random forests tends to stabilize after a part of the trees are grown [40]. In general, random forests can be useful to provide a baseline to compare with more sophisticated methods.

4.5 Support Vector Machines

The support vector Machine (SVM) classifies is a generalization of a linear classifier which finds are an optimal separation hyperplane that minimizes misclassifications [41]. 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[38]. The maximal margin is computed as the the smallest distance from the observations to the hyperplane. However, in general the data is not linearly separable. In this case, the SVM classifier allows some samples to be on the wrong side of hyperplane. SVM attempts to minimize the number of wrong classified samples, based on an optimization parameter that defines the cost of misclassification.

The fact SVMs have misclassified samples actually turns out to be an advantage, since such behavior makes the result robust to outliers and focus on getting a

correct result using most of the observations [38]. The solution for the hyperplane coefficients depends only on the samples that violates 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 widely applied to classify remote sensing data [25].

In `sits`, SVM is implemented as a wrapper of `e1071` R package that uses the `LIBSVM` implementation [42], 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 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).
.

```
# Filter the data slightly to reduce noise without reducing variability
samples_filtered <- sits_whittaker(samples_matogrosso_mod13q1,
                                   lambda = 0.5,
                                   bands_suffix = "")
# Train a machine learning model for the mato grosso dataset using SVM
# The parameters are those of the "e1071:svm" method
svm_model <- sits_train(samples_matogrosso_mod13q1,
                        ml_method = sits_svm())
# Classify using SVM model and plot the result
class <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.5, bands_suffix = "") %>%
    sits_classify(svm_model) %>%
    plot(bands = c("NDVI", "EVI"))
```

Compared with the output from the random forest classifier, the SVM result includes more classes. In particular, the transition from forest to pasture that occurs after 2004 is more gradual and shows a possible period of land abandonment in 2004 and 2005. Since the training dataset does not contain samples of deforested areas, places where forest is removed will tend to be classified as “Cerrado,” which is the nearest kind of vegetation cover where trees

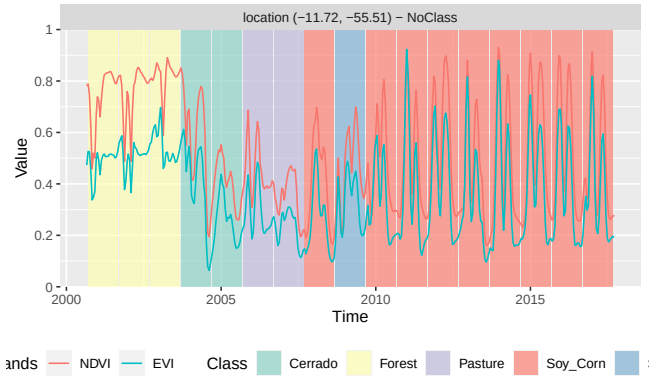


Figure 4.4: Classification of time series using SVM

and grasslands are mixed. Also, the classification for 2009 is different from the "Soy-Corn" label assigned from the other years from 2008 to 2017. In this example, the `sits_svm()` method is more sensitive to the distribution of the samples in the feature space than the random forests algorithm.

4.6 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 [40]. 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 [26], called XGBoost, is an efficient approximation to the gradient loss function. XGBoost is considered one of the best statistical learning algorithms available and has won many competitions; it is generally considered to be better than random forests. 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 conservative. 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.

```
# Train a machine learning model for the mato grosso dataset using XGBOOST
# The parameters are those of the "xgboost" package
xgb_model <- sits_train(samples_filtered, sits_xgboost(verbose = 0))
# Classify using SVM model and plot the result
point_mt_4bands %>%
    sits_whittaker(lambda = 0.50, bands_suffix = "") %>%
    sits_classify(xgb_model) %>%
    plot(bands = c("NDVI", "EVI"))
```

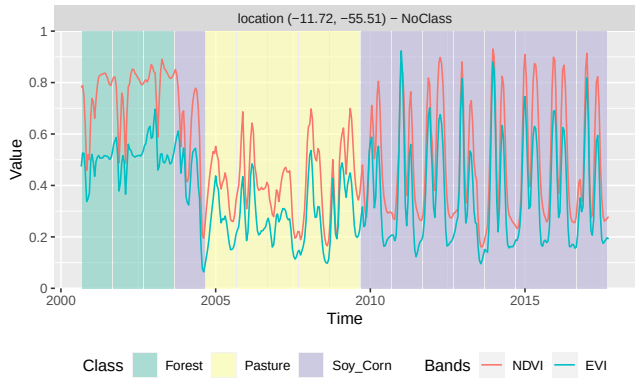


Figure 4.5: Classification of time series using XGBoost

In general, the results from the extreme gradient boosting model are similar to the Random Forest model. If desired, users can tune the hyperparameters need to perform using function `sits_kfold_validate()`. See Chapter 8 for more details.

4.7 Deep learning using multi-layer perceptrons

To support deep learning methods, `sits` uses the `keras` R package [43] as a backend. The first method is that of multi-layer perceptron (MLPs). These are the quintessential deep learning models. The goal of a multilayer perceptron is to approximate a function $y = f(x)$ that maps an input x to a category y. An MLP defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation [44]. An MLP consists of types 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. A user-defined number of 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 proven model for 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 (`activation`). The activation function of a node defines the output of that node given an input or set of inputs. Following standard practices [44], we recommend the use of the "relu" and "elu" functions.

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 [45]. Based on experience with image time series, we recommend that users start by using the default method provided by `sits`, which is the `optimizer_adam` method. Please refer to the `keras` package documentation for more information.

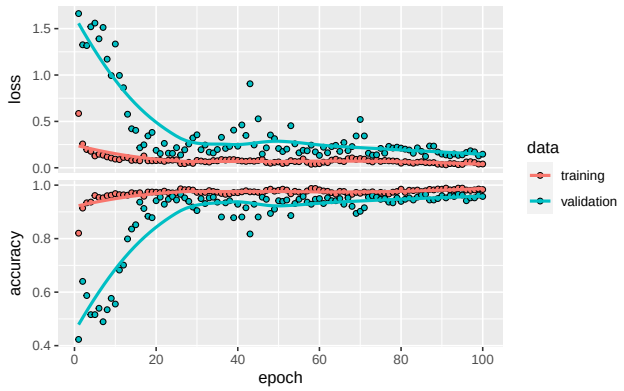
Another relevant parameter is the list of dropout rates (`dropout`). Dropout is a technique for randomly dropping units from the neural network during training [46]. By randomly discarding some neurons, dropout reduces overfitting. Since the purpose of a cascade of neural nets is to improve learning 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 [44]. 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 [47], which proposes the use of multi-layer 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 10%, 20%, and 30% for the layers; (d) the "optimizer_adam" 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.

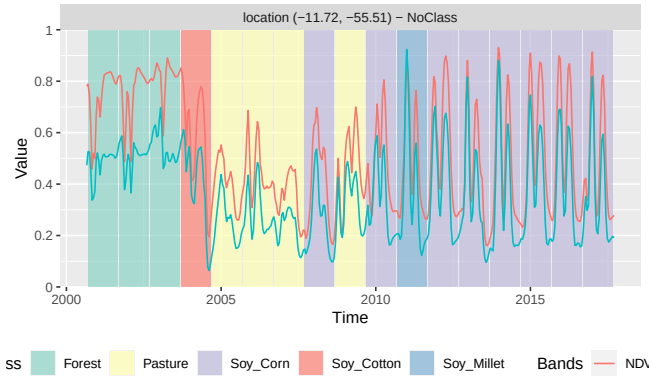
In our experience, if the training dataset is of good quality, using 3 to 5 layers is a reasonable compromise. Further increase on the number of layers will not improve the model. To simplify the output generation, the `verbose` option has been turned off. The default value is "on." After the model has 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 machine learning model for the Mato Grosso data using an MLP model

mlp_model <- sits_train(samples_filtered,
                        sits_mlp(
                        layers           = c(512, 512, 512),
                        activation       = "relu",
                        dropout_rates    = c(0.10, 0.20, 0.30),
                        epochs           = 100,
                        batch_size       = 64,
                        verbose          = 0,
                        validation_split = 0.2) )

# show training evolution
plot(mlp_model)
```



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

```
# Classify using DL model and plot the result
point_mt_6bands %>%
    sits_select(bands = c("NDVI", "EVI", "NIR", "MIR")) %>%
    sits_whittaker(lambda = 0.5, bands_suffix = "") %>%
    sits_classify(mlp_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



The multi-layer perceptron model is able to capture more subtle changes than the random forests and XGBoost models. For example, the transition from Forest to Pasture as estimated by the model is not abrupt and takes more than one year. In 2004, the time series corresponds to that of a degraded forest. Since there are no samples for "Forest Degradation," the model assigns this series to a class that is neither "Forest" nor "Pasture," which in our case is "Cerrado." This indicates that users should include samples of "Forest Degradation" to improve classification. 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.

4.8 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. [48] states that the use of 1D-CNN for time series classification improves on the use of multi-layer perceptrons, since the classifier is able to represent temporal relationships. 1D-CNNs with a suitable convolution window make 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 in [2]. The "TempCNN" architecture has three 1D convolutional layers (each with 64 units), one dense layer of 256 units and a final softmax layer for classification (see figure). The kernel size of the convolution filters is set to 5. 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 [2], the authors compare favourably the tempCNN model with the Recurrent Neural Network proposed by [29] for land use classification. The figure below shows the architecture of the tempCNN model.

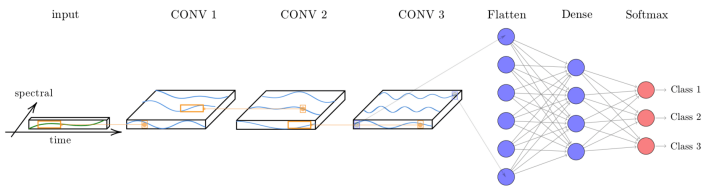
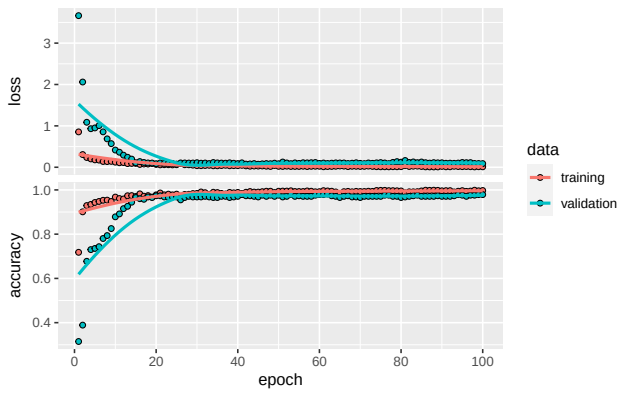


Figure 4.6: Structure of tempCNN architecture (source: Pelletier et al.(2019))

The function `sits_TempCNN()` implements the model. The code has been derived from the Python source provided by the authors (https://github.com/charlotte-pel/temporalCNN). Most of the parameters corresponds to those chosen by [2]. 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 64 units. The parameter `cnn_kernels` indicates the size of the convolution kernels; the default values are kernels of size 5. Activation for all 1D-CNN layers is set by `cnn_activation` (default = "relu"). The parameter `cnn_L2_rate` controls the regularization rate, with a default of "1e-06." The dropout rates for each 1D-CNN layer are controlled individually by the parameter `cnn_dropout_rates`.

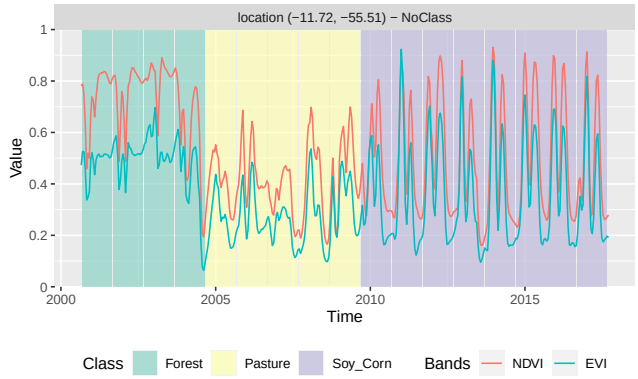
Based on discussions with the author, the defaults in `sits` range from 10% to 30%. The `validation_split` controls the size of the test set, relative to the full data set. We recommend to set aside at least 20% of the samples for validation.

```
# train a machine learning model using tempCNN
tCNN_model <- sits_train(samples_filtered,
                         sits_TempCNN(
                            cnn_layers           = c(64, 64, 64),
                            cnn_kernels          = c(5, 5, 5),
                            cnn_activation       = 'relu',
                            cnn_L2_rate          = 1e-06,
                            cnn_dropout_rates    = c(0.10, 0.20, 0.30),
                            epochs               = 100,
                            batch_size           = 64,
                            validation_split     = 0.2,
                            verbose              = 0) )

# show training evolution
plot(tCNN_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_whittaker(lambda = 0.5, bands_suffix = "") %>%
    sits_classify(tCNN_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



While the result of the TempCNN model using default parameters is similar to that of the MLP model, it has has the potential to better explore the time series data than the the MLP model. In our experience, TempCNN models are a reliable way for classifying image time series[49]. Recent work which compares different models also provides evidence of a satisfactory behavior[31].

4.9 Residual 1D CNN Networks (ResNet)

The Residual Network (ResNet) is a 1D convolution neural network (CNN) proposed by [47], based on the idea proposed by [50] for image recognition. 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[51]. Skip connections aim to avoid vanishing gradients from occurring, allowing deep networks to be trained.

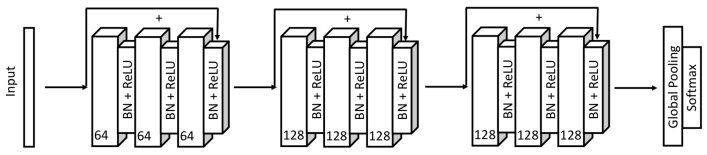
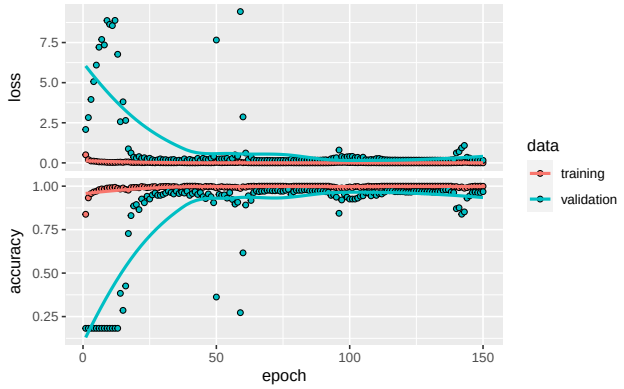


Figure 4.7: Structure of ResNet architecture (source: Wang et al.(2017))

In `sits`, the ResNet is implemented using the `sits_ResNet()` function. The default parameters are those proposed by [47], and we also benefited from the code provided by [32]. The first parameter is `blocks`, which controls the number

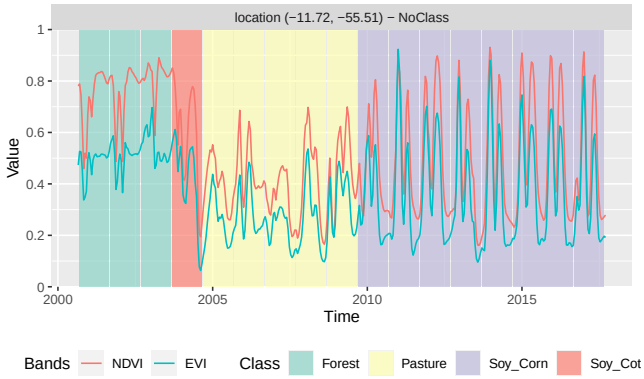
of blocks and the size of filters in each block. By default, the model implements three blocks, the first with 64 filters and the others with 128 filters. Users can control the number of blocks and filter size by changing this parameter. The parameter `kernels` controls the size the of kernels of the 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 the same as proposed in [47] and [32].

```
# train a machine learning model using ResNet
resnet_model <- sits_train(samples_filtered,
                        sits_ResNet(
                           blocks               = c(64, 128, 128),
                           kernels              = c(8, 5, 3),
                           activation           = 'relu',
                           epochs               = 150,
                           batch_size           = 64,
                           validation_split     = 0.2,
                           verbose              = 0) )
# show training evolution
plot(resnet_model)
```



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

```
# Classify using DL model and plot the result
point_mt_4bands <- sits_select(point_mt_6bands,
                               bands = c("NDVI", "EVI", "NIR", "MIR"))
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.5, bands_suffix = "") %>%
    sits_classify(resnet_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



4.10 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 [28], long short term memory convolutional networks (LSTM-FCN)[52] and temporal self-attention [30]. 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 [32] 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 available in comparison archives such as the UCR dataset.

In the specific case of satellite image time series, [31] presents 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 that 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. Given the small differences between the models, 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. Using clustering methods (Chapter 4) to improve sample quality is highly recommended.

Chapter 5

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.

5.1 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 parallel processing using the cores of the virtual machine. Users are not burdened with the need to learn how to do multiprocessing and, thus, their learning curve is shortened.

The authors implemented a new fault tolerant multitasking procedure for big EO data classification. 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.

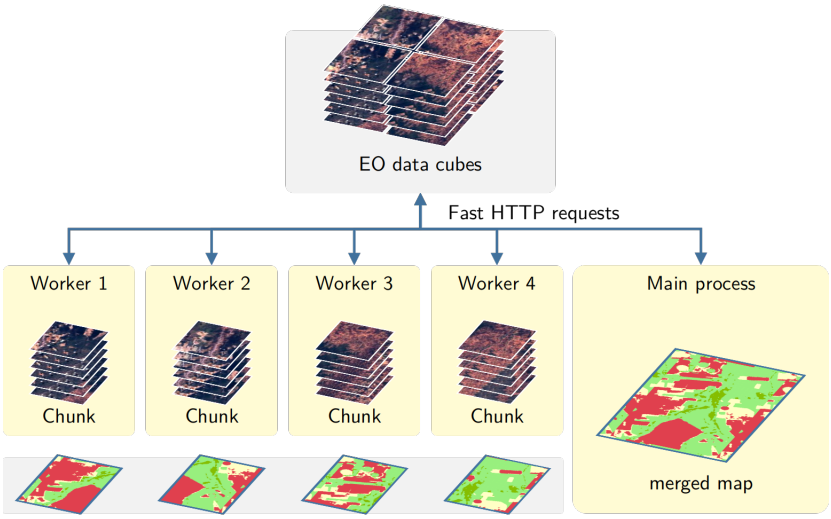


Figure 5.1: Parallel processing in sits (source: Simoes et al.(2021)

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 50% 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`. For small imagens, we suggest using a small number of cores (up to 4). In the case of big data, our experience is to use about 50% of the cores.

5.2 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 $km\hat{}2$) 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.

Part III

Post classification

Chapter 6

Post classification smoothing

This chapter describes the methods available for spatial smoothing of the results of machine learning classifications.

6.1 Introduction

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" [53]. 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" [54]: 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.

6.2 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 neighbouring 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.

6.2.1 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, $k = 1, \ldots, K$ at a time t, $t = 1, \ldots, T$. 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

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

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

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

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

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

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

$$[\mathbf{x}_{i,t}|\boldsymbol{\theta}_{i,t}] \sim \mathcal{N}_K(\boldsymbol{\theta}_{i,t}, \boldsymbol{\Sigma}_{i,t}),$$

where $\boldsymbol{\theta}_{i,t}$ is the mean parameter vector for the pixel i at time t, and $\boldsymbol{\Sigma}_{i,t}$ 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 $\boldsymbol{\Sigma}_{i,t}$. To

model our uncertainty about the parameter $\boldsymbol{\theta}_{i,t}$, 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.

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

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

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

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

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

The $\boldsymbol{\theta}_{i,t}$ 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 $P[\mathbf{x}_{i,t}|\boldsymbol{\theta}_{i,t}]$ represents the added information provided by our observation of $\mathbf{x}_{i,t}$. The *posterior* probability density function $P[\boldsymbol{\theta}_{i,t}|\mathbf{x}_{i,t}]$ 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 $\hat{\boldsymbol{\theta}}_{i,t}$ for the $\boldsymbol{\theta}_{i,t}|\mathbf{x}_{i,t}$ 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

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

The estimator value for the logit vector $\hat{\boldsymbol{\theta}}_{i,t}$ 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 $\boldsymbol{\Sigma}_{i,t}$ 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 $\boldsymbol{\Sigma}_{i,t}$, 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 $\boldsymbol{\Sigma}_{i,t}$, 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, $\boldsymbol{\Sigma}_{i,t}$ 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 $\boldsymbol{\Sigma}_{i,t}$ 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.

6.3 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(verbose = 0))

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 = "LOCAL",
                         origin     = "BDC",
                         collection = "MOD13Q1-6",
                         name = "Sinop",
                         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 = 1, multicores = 1,
                              output_dir = tempdir())

plot(raster_probs)
```

```
#> downsample set to c(3,3,1)
```

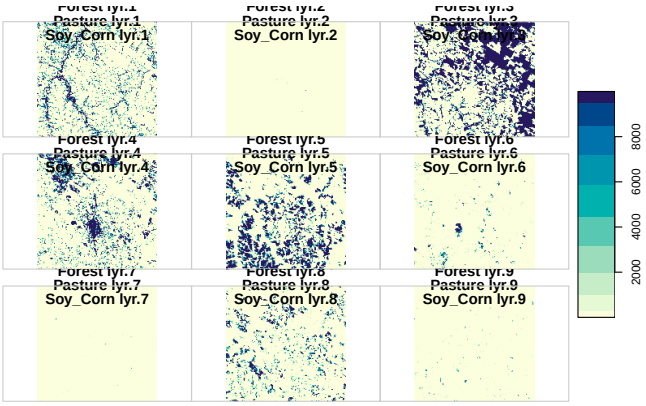


Figure 6.1: Probability values for classified image

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",
                                  output_dir = tempdir())
```

```
# plot the result
plot(raster_probs_bayes)
```

```
#> downsample set to c(3,3,1)
```

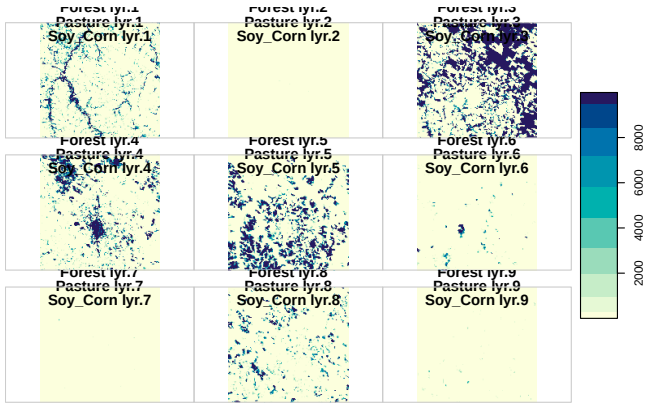


Figure 6.2: Probability values smoothed by bayesian method

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.

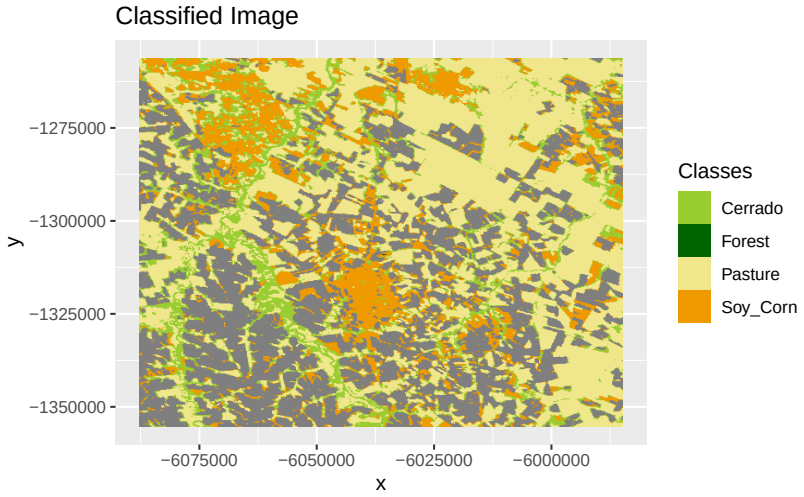


Figure 6.3: Classified image without smoothing

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

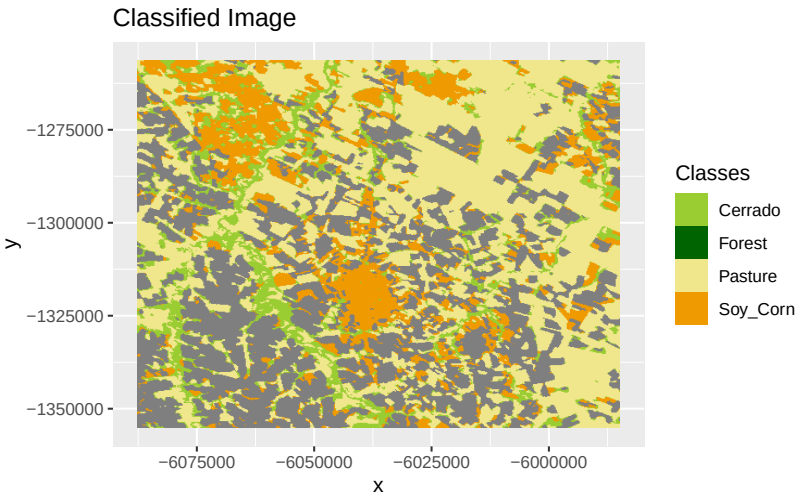


Figure 6.4: Classified image with Bayesian smoothing

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.

6.4 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 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 [54].

Bilateral smoothing combines proximity (combining pixels which are close) and similarity (comparing the values of the pixels) [55]. 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;
- $I(x_i)$ 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[56]. 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())
# plot the result
plot(raster_probs_bil)
```

```
#> downsample set to c(3,3,1)
```

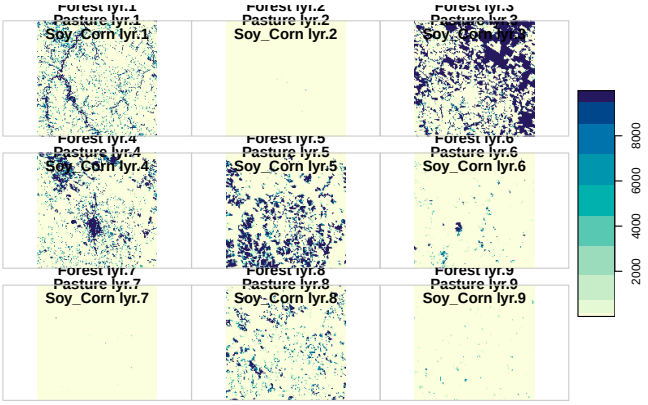


Figure 6.5: Probability values for classified image smoothed by bilateral filter

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

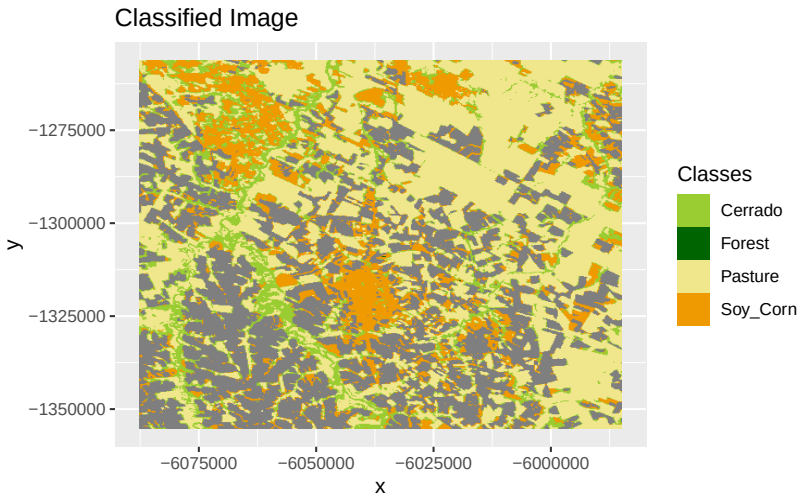


Figure 6.6: Classified image with bilateral smoothing

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 [54], smoothing improves the quality of classified images and thus should be applied in most situations.

Chapter 7

Validation and accuracy measurements in SITS

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

7.1 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. Here, 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 [40].

Cross-validation estimates the expected prediction error. It 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 [40], 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, which somewhat overestimates the true prediction error.

`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     392      15
#>    Pasture       8     331
#>
#>           Accuracy : 0.9692
#>             95% CI : (0.9541, 0.9804)
#>
#>              Kappa : 0.9379
#>
#>  Prod Acc  Cerrado : 0.9800
#>  Prod Acc  Pasture : 0.9566
#>  User Acc  Cerrado : 0.9631
#>  User Acc  Pasture : 0.9764
#>
```

7.2 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.

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

7.3 Accuracy assessment

7.3.1 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
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 : 1
#>    95% CI : (0.9951, 1)
#>
#>     Kappa : 1
```

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     400       0
#>    Pasture       0     346
#>
#>           Accuracy : 1
#>             95% CI : (0.9951, 1)
#>
#>              Kappa : 1
#>
```

```
#>  Prod Acc  Cerrado : 1
#>  Prod Acc  Pasture : 1
#>  User Acc  Cerrado : 1
#>  User Acc  Pasture : 1
#>
```

7.3.2 Classified images

To measure the accuracy of classified images, the `sits_accuracy` function uses an area-weighted technique, following the best practices proposed by [57]. 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, [57] 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 [57] and [58]. The following explanation is extracted from the paper of [57], 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 [57], 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 labeled points. The csv file should have the same format as the one used to obtain samples, as discussed earlier.

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_2bands <- sits_select(samples_modis_4bands,
                                    bands = c("NDVI", "EVI"))
# build an extreme gradient boosting model
xgb_model <- sits_train(samples_modis_2bands, ml_method = sits_xgboost())
# create a data cube based on files
data_dir <- system.file("extdata/raster/mod13q1", package = "sits")
cube <- sits_cube(
    source     = "LOCAL",
    origin     = "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())
# 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 = 0.765\begin{table}
#>
#> \caption{\label{tab:unnamed-chunk-79}Area-Weighted Users and Producers Accuracy}
#> \centering
#> \begin{tabular}[t]{l|r|r}
#> \hline
#>   & User & Producer\\
#> \hline
#> Cerrado & 1.00 & 0.67\\
#> \hline
#> Forest & 0.60 & 1.00\\
#> \hline
#> Pasture & 0.75 & 0.61\\
```

```
#> \hline
#> Soy\_Corn & 0.86 & 0.72\\
#> \hline
#> \end{tabular}
#> \end{table}
#> \begin{table}
#>
#> \caption{\label{tab:unnamed-chunk-79}Mapped Area x Estimated Area (ha)}
#> \centering
#> \begin{tabular}[t]{l|r|r|r}
#> \hline
#>   & Mapped Area (ha) & Error-Adjusted Area (ha) & Conf Interval (ha)\\
#> \hline
#> Cerrado & 32569.0 & 48736.0 & 31687.3\\
#> \hline
#> Forest & 80834.8 & 48500.9 & 38808.8\\
#> \hline
#> Pasture & 19029.4 & 23393.5 & 20163.5\\
#> \hline
#> Soy\_Corn & 63850.0 & 75652.9 & 37558.6\\
#> \hline
#> \end{tabular}
#> \end{table}
```

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 [58] to estimate the number of ground truth measures per class that are required to get a reliable estimate.

Chapter 8

Case studies

This chapter presents case studies using sits

Chapter 9

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.

9.1 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 [59], Open Data Cube [60] and openEO [61]. In what follows, we compare **sits** to each of these solutions.

Google Earth Engine (GEE) [59] uses the Google distributed file system [62] and its map-reduce paradigm [63]. By combining a flexible API with an efficient back-end processing, GEE has become a widely used platform [64]. 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 [65]. 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 [61] 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] K. Didan, "MOD13Q1 MODIS/Terra Vegetation Indices 16-Day L3 Global 250m SIN Grid V006," NASA EOSDIS Land Processes DAAC, 2015.

[2] 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.

[3] 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.

[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] H. H. Madden, "Comments on the Savitzky-Golay convolution method for least-squares-fit smoothing and differentiation of digital data," *Analytical chemistry*, vol. 50, no. 9, pp. 1383–1386, 1978.

[9] 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 SavitzkyGolay filter," *Remote Sensing of Environment*, vol. 91, no. 3, pp. 332–344, 2004, doi: 10.1016/j.rse.2004.03.014.

[10] 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.

[11] E. T. Whittaker, "On a new method of graduation," *Proceedings of the Edinburgh Mathematical Society*, vol. 41, pp. 63–75, 1922.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] C. Hennig, "Clustering strategy and method selection," in *Handbook of cluster analysis*, C. Hennig, M. Meila, F. Murtagh, and R. Rocci, Eds. CRC Press, 2015.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[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.

[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.

[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.

[36] 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.

[37] 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.

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

[39] 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.

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

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

[42] 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.

[43] F. Chollet and J. J. Allaire, *Deep Learning with R*. Manning Publications Co., 2018.

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

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

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] F. Karim, S. Majumdar, H. Darabi, and S. Harford, "Multivariate LSTM-FCNs for time series classification," *Neural Networks*, vol. 116, pp. 237–245, 2019, doi: 10.1016/j.neunet.2019.04.014.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] M. Schramm *et al.*, "The openEO APIHarmonising 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.

[62] 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.

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

[64] A. Ghorbanian, M. Kakooei, M. Amani, S. Mahdavi, A. Mohammadzadeh, and M. Hasanlou, "Improved land cover map of iran using sentinel imagery within google earth engine and a novel automatic workflow for land cover classification using migrated training samples," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 167, pp. 276–288, 2020.

[65] 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.