sits: Data Analysis and Machine Learning for Data Cubes using Satellite Image Time Series

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

2021-03-31

Contents

Preface

Abstract from Filters: This vignette describes the time series filtering techniques available in the SITS package. These include the Savitsky-Golay, Whittaker, and Kalman filters, as well as specialised filters for satellite images, which are the envelope filter and the ARIMA filter for cloud removal.

Abstract from clustering: 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. Representative samples lead to good classification maps. `sits` provides support for two clustering methods to test sample quality, which are agglomerative hierarchical clustering (AHC) and self-organizing maps (SOM).

Abstract from clustering: 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. Representative samples lead to good classification maps. `sits` provides support for two clustering methods to test sample quality, which are agglomerative hierarchical clustering (AHC) and self-organizing maps (SOM).

Chapter 1

Introduction

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. Here, we present an open-source *R* package for satellite image time series analysis called **sits**. The package support use of machine learning techniques for classification image time series obtained from data cubes. Methods available include linear and quadratic discrimination analysis, support vector machines, random forests, boosting, deep learning, and convolution neural.

Earth observation 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?*

The approach taken in the current work is to develop data analysis methods that work with satellite image time series, obtained by taking calibrated and comparable measures of the same location in Earth at different times. These measures can be obtained by a single sensor (e.g., MODIS) or by combining various sensors (e.g., Landsat 8 and Sentinel-2). If acquired by frequent revisits, these data sets' temporal resolution can capture significant land use changes.

Time series of remote sensing data show that land cover can occur not only progressively and gradually, but they may also show discontinuities with abrupt changes (Lambin, Geist, and Lepers 2003). Analyses of multiyear time series

of land surface attributes, their fine-scale spatial pattern, and their seasonal evolution lead to a broader view of land-cover change. Satellite image time series have already been used in applications such as mapping for detecting forest disturbance (Kennedy, Yang, and Cohen 2010), ecology dynamics (Pasquarella et al. 2016), agricultural intensification (Galford et al. 2008), and its impacts on deforestation (Arvor et al. 2012). Algorithms for processing image time series include BFAST for detecting breaks (Verbesselt et al. 2010), TIMESAT for modeling and measuring phenological attributes (Jönsson and Eklundh 2004), and methods based on Dynamic Time Warping (DTW) for land use and land cover classification (Petitjean, Inglada, and Gancarski 2012)(Maus et al. 2016).

In this work, 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. These methods include linear and quadratic discrimination analysis, support vector machines, random forests, and neural networks. One important contribution of the package is to support the complete cycle of data analysis for time series classification, including data acquisition, visualization, filtering, clustering, classification, validation, and post-classification adjustments.

Most studies using satellite image time series for land cover classification use a *space-first, time-later* approach. For multiyear studies, researchers first derive best-fit yearly composites and then classify each composite image. To review these methods for land use and land cover classification using time series, see (Gomez, White, and Wulder 2016). As an alternative to *space-first, time-later* methods, the **sits** package provides support for the classification of time series, preserving the full temporal resolution of the input data, using a *time-first, space-later* approach. **sits** uses all data in the image time series to create larger dimensional spaces for machine learning. The idea is to have as many temporal attributes as possible, increasing the classification space's dimension. Each temporal instance of a time series is taken as an independent dimension in the classifier's feature space. To the authors' best knowledge, the classification techniques for image time series included in the package are not previously available in other R or python packages. Furthermore, the package provides filtering, clustering, and post-processing methods that have not been published in the literature.

1.1 Workflow

The main aim of **sits** is to support land cover and land change classification of image data cubes using machine learning methods. The basic workflow is:

1. Create a data cube using image collections available in the cloud or local machines.
2. Extract time series from the data cube, which is used as training data.
3. Perform quality control and filtering on the samples.
4. Train a machine learning model using the extracted samples.

5. Classify the data cube using the trained model.
6. Post-process the classified images.

1.2 Handling Data Cubes in sits

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

In broad terms, the cloud computing model is one where large satellite-generated data sets are archived on cloud services, providing computing facilities to process them. By using cloud services, users can share big Earth observation databases and minimize data download. Investment in infrastructure is minimized, and sharing of data and software increases. However, data available in the cloud is best organised for analysis by creating data cubes.

Generalizing Appel and Pebesma (2019), we consider that a data cube is a four-dimensional structure with dimensions x (longitude or easting), y (latitude or northing), time, and bands. Its spatial dimensions refer to a single spatial reference system (SRS). Cells of a data cube have a constant spatial size (concerning the cube's SRS). A set of intervals specifies the temporal dimension. For every combination of dimensions, a cell has a single value. Data cubes are particularly amenable for machine learning techniques; their data can be transformed into arrays in memory, fed to training and classification algorithms. Given the widespread availability of large data sets of Earth observation data, there is a growing interest in organising large data sets into "data cubes".

1.2.2 Using STAC to Access Image Data Cubes

One of the distinguishing features of **sits** is that it has been designed to work with big satellite image data sets which reside on the cloud and with data cubes. Many *R* packages working with remote sensing images require data to be accessible in a local computer. However, with the coming of age of big Earth observation data, it is not always practical to transfer large data sets. Users have to rely on web services to provide access to these data sets. In this context, **sits** is based on access to data cubes using the information provided by STAC (Spatio-temporal Access Catalogue).

Currently, **sits** supports data cubes available in the following cloud services:

1. Sentinel-2/2A level 2A images in Amazon Web Services (AWS);
2. Collections of Sentinel, Landsat, and CBERS images in the Brazil Data Cube (BDC);
3. Collections available in Digital Earth Africa;
4. Data cubes produced by the gdalcubes package;
5. Local image collections.

In order, to access different STAC cloud services supported by **sits**, the rstac

package is used. This package is developed under the Brazil Data Cube project and provides features for consuming STAC services. Besides, the aws.s3 package is used to access data in AWS.

The use of different providers in the **sits** package is done with as few change as possible for users. The user can define a data cube by selecting a cloud service collection and determining a space-time extent. The code below shows the definition of a data cube using AWS Sentinel-2/2A images to exemplify how it is used.

```
s2_cube <- sits_cube(
    source        = "AWS",
    name          = "T20LKP_2018_2019",
    collection    = "sentinel-s2-l2a",
    bands         = c("B08", "SCL"),
    tiles         = "20LKP",
    start_date    = as.Date("2018-07-18"),
    end_date      = as.Date("2018-08-18"),
    s2_resolution = 60
)
```

In the above example, the user has selected the "Sentinel-2 Level 2" collection in the AWS cloud services. The data cube's geographical area is defined by the tile "20LKP" and the temporal extent by a start and end date. Access to other cloud services works in similar ways.

Users can derive data cubes from ARD data that have pre-defined temporal resolutions. For example, a user may want to define the best Sentinel-2 pixel in a one-month period, as shown below. This can be done in **sits** by the `sits_regularize`, which calls the "gdalcubes" package. For details in gdalcubes, please see https://github.com/appelmar/gdalcubes.

```
gc_cube <- sits_regularize(
          cube       = s2_cube,
          name       = "T20LKP_2018_2019_1M",
          dir_images = tempdir(),
          period     = "P1M",
          agg_method = "median",
          resampling = "bilinear",
          cloud_mask = TRUE
)
```

1.2.3 Defining a data cube using files

To define a data cube using plain files (without STAC information), all image files should have the same spatial resolution and same projection. Each file contains a single image band for a single date. Since raster files in popular formats (e.g., GeoTiff and JPEG 2000) do not include time information, each

file's name needs to include the date and band information. Timeline and bands are deduced from filenames. For example, `CBERS-4_AWFI_B13_2018-02-02.tif` is a valid name. The user has to provide parsing information to allow **sits** to extract the band and the date. In the example above, the parsing info is `c("X1", "X2", "band", "date")` and the delimiter is `"_"`.

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

# files are named using the convention
# "CBERS-4_AWFI_B13_2018-02-02.tif"
cbers_cube <- sits_cube(
      source = "LOCAL",
      name = "022024",
      satellite = "CBERS-4",
      sensor = "AWFI",
      data_dir = data_dir,
      delim = "_",
      parse_info = c("X1", "X2", "band", "date")
)

# print the timeline of the cube
sits_timeline(cbers_cube)
```

```
#>  [1] "2018-02-02" "2018-02-18" "2018-03-06" "2018-03-22" "2018-04-07"
#>  [6] "2018-04-23" "2018-05-09" "2018-05-25" "2018-06-10" "2018-06-26"
#> [11] "2018-07-12" "2018-07-28" "2018-08-13" "2018-08-29"
```

```
# print the bounding box of the cube
sits_bbox(cbers_cube)
```

```
#>    xmin    xmax    ymin    ymax
#> 5794837 5798037 9773148 9776348
```

1.3 Handling satellite image time series in sits

1.3.1 Data structure

Training a machine learning model in **sits** requires a set of time series describing properties in spatio-temporal locations of interest. This set consists of samples provided by experts that take *in-situ* field observations or recognize land classes using high-resolution images for land use classification. 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.

The package uses a `sits tibble` to organize time series data with associated

spatial information for handling time series. A `tibble` is a generalization of a `data.frame`, the usual way in *R* to organize data in tables. Tibbles is part of the `tidyverse`, a collection of R packages designed to work together in data manipulation (Wickham and Grolemund 2017). As an example of how the **sits** tibble works, the following code shows the first three lines of a tibble containing 1,218 labeled samples of land cover in the Mato Grosso state of Brazil, with four classes: "Forest", "Cerrado", "Pasture", "Soybean-Corn".

```
# data set of samples
data("samples_modis_4bands")
samples_modis_4bands[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 "Pasture" at location (−55.1852, −10.8378) and is valid for the period (2013-09-14, 2014-08-29).

```
# print the first time series records of the first sample
sits_time_series(samples_modis_4bands[1,])[1:3,]
```

```
#> # A tibble: 3 x 5
#>   Index       NDVI   EVI   NIR   MIR
#>   <date>     <dbl> <dbl> <dbl> <dbl>
#> 1 2013-09-14 0.388 0.253 0.316 0.307
#> 2 2013-09-30 0.491 0.277 0.275 0.170
#> 3 2013-10-16 0.527 0.318 0.286 0.205
```

1.3.2 Obtaining time series data

To get a time series in **sits**, first is must necessarily create a data cube. Users can request one or more time series points from a data cube using `sits_get_data()`. This function provides a general means of access to image time series. Given data cue, 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()`.

```
library(sits)
data_dir <- system.file("extdata/raster/mod13q1", package = "sits")
```

```
modis_cube <- sits_cube(
    source = "LOCAL",
    name = "sinop-2014",
    satellite = "TERRA",
    sensor = "MODIS",
    data_dir = data_dir,
    delim = "_",
    parse_info = c("X1", "X2", "band", "date")
)

# obtain a set of locations defined by a CSV file
csv_raster_file <- system.file("extdata/samples/samples_sinop_crop.csv",
                               package = "sits")

# retrieve the points from the data cube
points <- sits_get_data(modis_cube, file = csv_raster_file)
```

```
#> All points have been retrieved
```

```
# plot the first point
plot(points[1,])
```

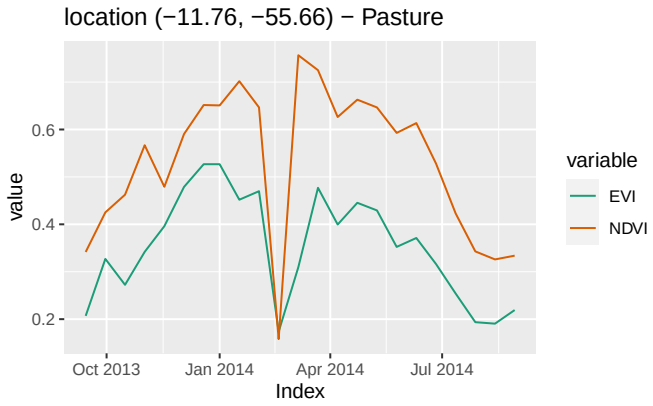


Figure 1.1: A one year time series of MOD13Q1 data for bands NDVI and EVI

1.4 Filtering techniques

The literature on satellite image time series has several filtering applications to correct or smooth vegetation index data. The following filters are available in **sits**. They are described in more detail in the vignette "Satellite Image Time Series Filtering with SITS":

- Savitzky–Golay filter (`sits_sgolay`)

- Whittaker filter (`sits_whittaker`)
- Envelope filter (`sits_envelope`)

The **sits** package uses a common interface to all filter functions with the `sits_filter`. The function has two parameters: the `dataset` to be filtered and the `filter` function applied. To aid in data visualization, all filtered bands have a suffix that is appended, as shown in the examples below. Here we show an example using the Whittaker smoother, proposed in the literature (Atzberger and Eilers 2011) as arguably the most appropriate one to use for satellite image time series. The Whittaker smoother attempts to fit a curve representing the raw data but is penalized if subsequent points vary too much (Atzberger and Eilers 2011). It balances between the residual to the original data and the "smoothness" of the fitted curve. It uses the parameter `lambda` to control the degree of smoothing.

```
# Take a NDVI time series, apply Whittaker filter and plot the series
point_ndvi <- sits_select(point_mt_6bands, bands = "NDVI")
point_whit <- sits_whittaker(point_ndvi, lambda = 5.0)

# merge with original data and plot the original and the filtered data
point_whit %>%
  sits_merge(point_ndvi) %>%
  plot()
```

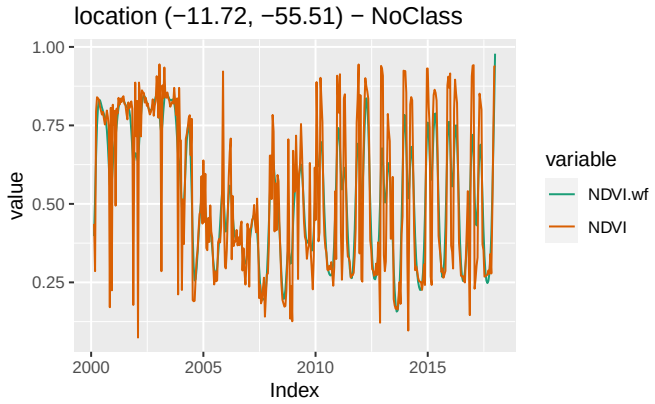


Figure 1.2: Whittaker smoother filter applied on one-year NDVI time series. The example uses default $\lambda = 3$ parameter.

1.5 Clustering for sample quality control using self-organizing maps

One of the key challenges of machine learning classification models is assessing the training data sets' quality. It helps apply pre-processing methods to improve

the samples' quality and remove those that might have been wrongly labeled or have low discriminatory power. Good samples lead to good classification maps. **sits** provides support for two clustering methods to test sample quality: (a) Agglomerative Hierarchical Clustering (AHC); (b) Self-organizing Maps (SOM). Full details of the cluster methods used in **sits** are available in the vignette 'Clustering of Satellite Image Time Series with SITS'.

1.6 Classification using machine learning

There has been much recent interest in using classifiers such as support vector machines (Mountrakis, Im, and Ogole 2011) and random forests (Belgiu and Dragut 2016) for remote sensing images. Most often, researchers use a *space-first, time-later* approach. The dimension of the decision space is limited to the number of spectral bands or their transformations. Sometimes, the decision space is extended with temporal attributes. To do this, researchers filter the raw data to get smoother time series (Brown et al. 2013; Kastens et al. 2017). Using software such as TIMESAT (Jönsson and Eklundh 2004), they derive a small set of phenological parameters from vegetation indexes, like the beginning, peak, and length of the growing season (Estel et al. 2015; Pelletier et al. 2016).

In a recent review of machine learning methods to classify remote sensing data (Maxwell, Warner, and Fang 2018), the authors note that many factors influence these classifiers' performance, including the size and quality of the training dataset dimension of the feature space and the choice of the parameters. We support both *space-first, time-later* and *time-first, space-later* approaches. Therefore, the **sits** package provides functionality to explore the full depth of satellite image time series data.

When used in *time-first, space-later* approach, **sits** treats time series as a feature vector. To be consistent, the procedure aligns all-time series from different years by its time proximity considering a given cropping schedule. Once aligned, the feature vector is formed by all pixel "bands". The idea is to have as many temporal attributes as possible, increasing the classification space's dimension. In this scenario, statistical learning models are the natural candidates to deal with high-dimensional data: learning to distinguish all land cover and land use classes from trusted samples exemplars (the training data) to infer classes of a larger data set.

The **sits** package provides a common interface to all machine learning models, using the `sits_train` function. This function takes two parameters: the input `data samples` and the ML method (`ml_method`), as shown below. After the model is estimated, it can classify individual time series or full data cubes using the `sits_classify` function. In the examples that follow, we show how to apply each method to classify a single time series. Then, we discuss how to organize full data cubes.

When a dataset of time series organised as a 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 following example illustrates how to train a dataset and classify an individual time series. First, we use the `sits_train` function with two parameters: the training dataset (described above) and the chosen machine learning model (in this case, a random forest classifier). The trained model is then used to classify a time series from Mato Grosso Brazilian state, using `sits_classify`. The results can be shown in text format using the function `sits_show_prediction` or graphically using the `plot` function.

```
#select the data for classification

# Train a machine learning model using Random Forest
rfor_model <- sits_train(data = samples_modis_4bands,
                    ml_method = sits_rfor(num_trees = 1000))

# get a point to be classified
point_4bands <- sits_select(point_mt_6bands,
                            bands = c("NDVI", "EVI", "NIR", "MIR"))

# Classify using random forest model and plot the result
class.tb <- sits_classify(point_4bands, rfor_model)
# show the results of the prediction
sits_show_prediction(class.tb)
```

```
#> # 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
```

```
# plot the results of the prediction
plot(class.tb)
```

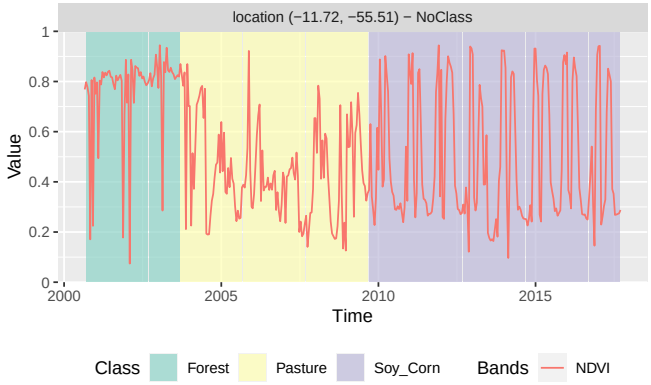


Figure 1.3: Random forest classification of a 16 years time series. The location (latitude, longitude) shown at the top of the graph is in geographic coordinate system (WGS84 *datum*).

The following methods are available in **sits** for training machine learning models:

- Linear discriminant analysis (`sits_lda`)
- Quadratic discriminant analysis (`sits_qda`)
- Multinomial logit and its variants 'lasso' and 'ridge' (`sits_mlr`)
- Support vector machines (`sits_svm`)
- Random forests (`sits_rfor`)
- Extreme gradient boosting (`sits_xgboost`)
- Deep learning (DL) using multi-layer perceptrons (`sits_deeplearning`)
- DL with 1D convolutional neural networks (`sits_CNN`),
- DL combining 1D CNN and multi-layer perceptron networks (`sits_tempCNN`)
- DL using 1D version of ResNet (`sits_ResNet`).
- DL using a combination of long-short term memory (LSTM) and 1D CNN (`sits_LSTM_FCN`)

For more details on each method, please see the vignette "Machine Learning for Data Cubes using the sits package"

1.7 Validation techniques

Validation is a process undertaken on models to estimate some errors associated with them. Hence, it has been used widely in different scientific disciplines. Here, we are interested in assessing the prediction error associated with some models. For this purpose, we concentrate on the *cross-validation* approach, probably the most used validation technique (Hastie, Tibshirani, and J. 2009).

To be sure, 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. In 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 the 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 Hastie, Tibshirani, and J. (2009), there is a bias-variance trade-off in the 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 computationally expensive as it requires the same number of a 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 (Hastie, Tibshirani, and J. 2009), which somewhat overestimates the true prediction error.

`sits_kfold_validate()` gives support the k-fold validation in **sits**. The following code gives an example of how to proceed *k-fold cross-validation* in the package. It performs a five-fold validation using the 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
acc <- sits_kfold_validate(cerrado_2classes,
                           folds = 5,
                           ml_method = sits_rfor(num_trees = 1000))
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction Cerrado Pasture
#>    Cerrado     393      15
#>    Pasture       7     331
#>
#>           Accuracy : 0.9705
#>             95% CI : (0.9557, 0.9814)
#>
#>              Kappa : 0.9406
#>
#>  Prod Acc  Cerrado : 0.9825
#>  Prod Acc  Pasture : 0.9566
#>  User Acc  Cerrado : 0.9632
```

```
#>  User Acc  Pasture : 0.9793
#>
```

1.8 Cube classification

The continuous observation of the Earth's surface provided by orbital sensors is unprecedented in history. Just for the sake of illustration, a unique tile from MOD13Q1 product, a square of 4800 pixels provided every 16 days since February 2000, takes around 18GB of uncompressed data to store only one band or vegetation index. This data deluge puts the field into a big data era. It imposes challenges to design and build technologies that allow the Earth observation community to analyze those data sets (Câmara et al. 2017).

To classify a data cube, use the function `sits_classify()` as described below. This function works with data cubes built with the classification algorithms. It allows users to choose how many processes will run the task in parallel and the size of each data chunk consumed at each iteration. This strategy enables **sits** to work on average desktop computers without depleting all computational resources. The code below illustrates how to classify a small raster brick image that accompanies the package.

1.8.1 Steps for cube classification

Once a data cube that has associated files is defined, the steps for classification are:

1. Select a set of training samples;
2. Train a machine learning model;
3. Classify the data cubes using the model, producing a data cube with class probabilities;
4. Label the cube with probabilities, including data smoothing if desired.

1.8.2 Adjustments for improved performance

To reduce processing time, it is necessary to adjust `sits_classify()` according to the server's capabilities. The package tries to keep memory use to a minimum, performing garbage collection to free memory as often as possible. Nevertheless, there is an inevitable trade-off between computing time, memory use, and I/O operations. The best trade-off must be determined by the user, considering disk read speed, number of cores in the server, and CPU performance.

The first parameter is `memsize`. It controls the size of the main memory (in GBytes) to be used for classification. The user must specify how much free memory will be available. The second factor controlling the performance of raster classification is `multicores`. Once a block of data is read from the disk into the main memory, it is split into different cores, as specified by the user. In general, the more cores are assigned to classification, the faster the result will

be. However, there are overheads in switching time, especially when the server has other processes running.

Based on recent experience, the classification of a Sentinel-2 tile at 20-meter resolution (5490*5490 pixels), with six bands and 66-time instances covering one year of data, using SVM with a training data set of about 10,000 samples, takes about 3 hours using 20 cores and a memory size of 60 GB, in a server with 2.4GHz Xeon CPU and 96 GB of memory to produce the yearly classification maps.

```
# Retrieve the set of samples for the Mato Grosso region
# Select the data for classification
samples_2bands  <- sits_select(samples_modis_4bands,
                               bands = c("NDVI", "EVI"))

# build a machine learning model for this area
svm_model <- sits_train(samples_2bands, sits_svm())

# create a data cube to be classified
# Cube is composed of MOD13Q1 images from the Sinop region in Mato Grosso (Brazil)
data_dir <- system.file("extdata/raster/mod13q1", package = "sits")
sinop <- sits_cube(
    source = "LOCAL",
    name = "sinop-2014",
    satellite = "TERRA",
    sensor = "MODIS",
    data_dir = data_dir,
    delim = "_",
    parse_info = c("X1", "X2", "band", "date")
)
# Classify the raster cube, generating a probability file
probs_cube <- sits_classify(sinop,
                            ml_model = svm_model,
                            output_dir = tempdir(),
                            memsize = 16,
                            multicores = 4,
                            verbose = FALSE)

# plot the probabilities cubes
plot(probs_cube)
```

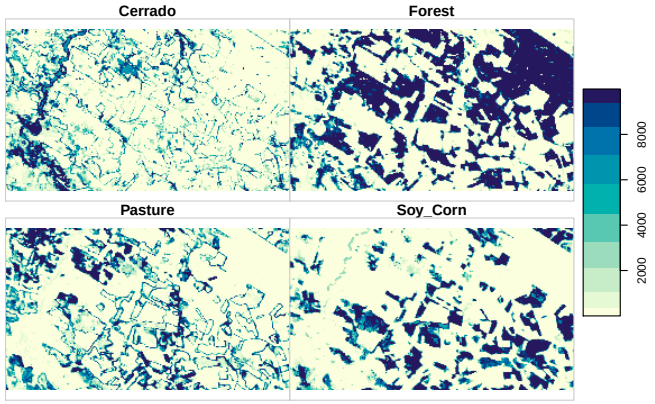


Figure 1.4: Class probabilities for each pixel

1.9 Smoothing and Labelling of raster data after classification

Post-processing is a desirable step in any classification process. Most statistical classifiers use training samples derived from "pure" pixels. Users have selected 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" (Schindler 2012): nearby pixels tend to have the same label. To put this assumption in practice, smoothing methods use the neighborhood information to remove outliers and enhance consistency in the resulting product.

Smoothing methods are an essential complement to machine learning algorithms for image classification. Since these methods are primarily pixel-based, they complement them with post-processing smoothing to include spatial information in the result. For each pixel, machine learning and other statistical algorithms provide the pixel's probabilities 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. **sits** uses a Bayesian smoothing method, which provides the means to incorporate prior knowledge in data analysis. Please see the vignette "Post classification smoothing using Bayesian techniques in SITS" for more details on the smoothing procedure.

Doing post-processing using Bayesian smoothing in **sits** is straightforward. The result of the `sits_classify` function applied to a data cube is a set of more probability images, one per requested classification interval. The next step is to use the `sits_smooth` function. By default, this function selects the most

likely class for each pixel, considering a Bayesian estimator that considers the neighbors. The following example takes the previously produced classification output and applies a Bayesian smoothing.

```
# smooth the result with a bayesian filter
sinop_bayes <- sits_smooth(probs_cube, output_dir = tempdir())
# label the resulting image
label_bayes <- sits_label_classification(sinop_bayes, output_dir = tempdir())
```

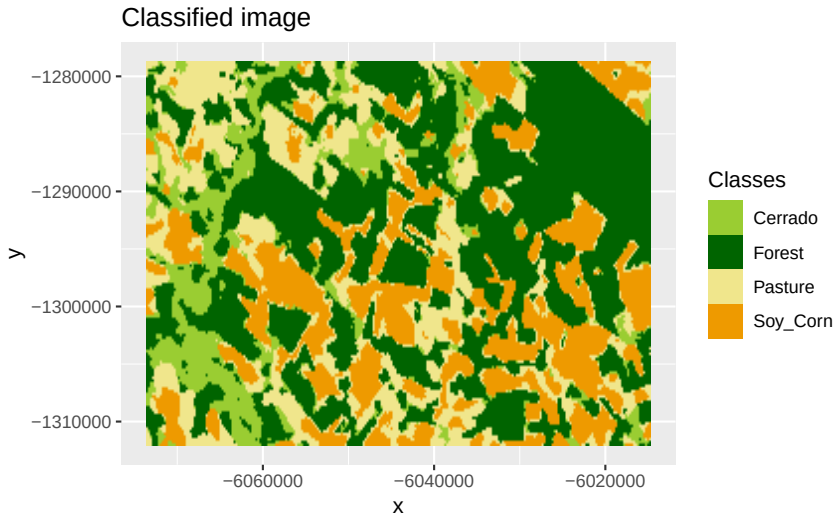


Figure 1.5: Classified image post-processed with Bayesian smoothing

1.10 Final remarks

Current approaches to image time series analysis still use a limited number of attributes. A common approach is deriving a small set of phenological parameters from vegetation indices, like the beginning, peak, and length of growing season (Brown et al. 2013), (Kastens et al. 2017), (Estel et al. 2015), (Pelletier et al. 2016). These phenological parameters are then fed in specialized classifiers such as TIMESAT (Jönsson and Eklundh 2004). These approaches do not use the power of advanced statistical learning techniques to work on high-dimensional spaces with big training data sets (James et al. 2013).

Package **sits** can use the full depth of satellite image time series to create larger dimensional spaces. We tested different methods of extracting attributes from time series data, including those reported by Pelletier et al. (2016) and Kastens et al. (2017). We conclude that part of the information in the raw time series is lost after filtering. Thus, the method we developed uses all the data available in the time series samples. The idea is to have as many temporal attributes as possible, increasing the classification space's dimension. Our experiments found

that modern statistical models such as support vector machines and random forests perform better in high-dimensional spaces than in lower-dimensional ones.

1.11 Acknowledgements

The authors would like to thank all the researchers that provided data samples used in the examples: Alexandre Coutinho, Julio Esquerdo, and Joao Antunes (Brazilian Agricultural Research Agency, Brazil) who provided ground samples for "soybean-fallow", "fallow-cotton", "soybean-cotton", "soybean-corn", "soybean-millet", "soybean-sunflower", and "pasture" classes; Rodrigo Bergotti (National Institute for Space Research, Brazil) who provided samples for "cerrado" and "forest" classes; and Damien Arvor (Rennes University, France) who provided ground samples for "soybean-fallow" class.

This work was partially funded by the São Paulo Research Foundation (FAPESP) through the eScience Program grant 2014/08398-6. We thank the Coordination for the Improvement of Higher Education Personnel (CAPES) and National Council for Scientific and Technological Development (CNPq) grants 312151/2014-4 (GC) and 140684/2016-6 (RS). We thank Ricardo Cartaxo and Lúbia Vinhas, who provided insight and expertise to support this paper.

This work has also been supported by the International Climate Initiative of the Germany Federal Ministry for the Environment, Nature Conservation, Building and Nuclear Safety under Grant Agreement 17-III-084-Global-A-RESTORE+ ("RESTORE+: Addressing Landscape Restoration on Degraded Land in Indonesia and Brazil").

Chapter 2

Acessing time series information in SITS

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 spatio-temporal 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 organise data in tables. Tibbles are part of the `tidyverse`, a collection of R packages designed to work together in data manipulation (Wickham and Grolemund 2017). As an example of how the `sits` tibble works, the following code shows the first three lines of a tibble containing $1,882$ labelled 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 spatial 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
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 labelled "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 **sits** 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` (Wickham and Grolemund 2017) 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 15 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 15 samples
plot(samples_cerrado[1:15,])
```

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)
```

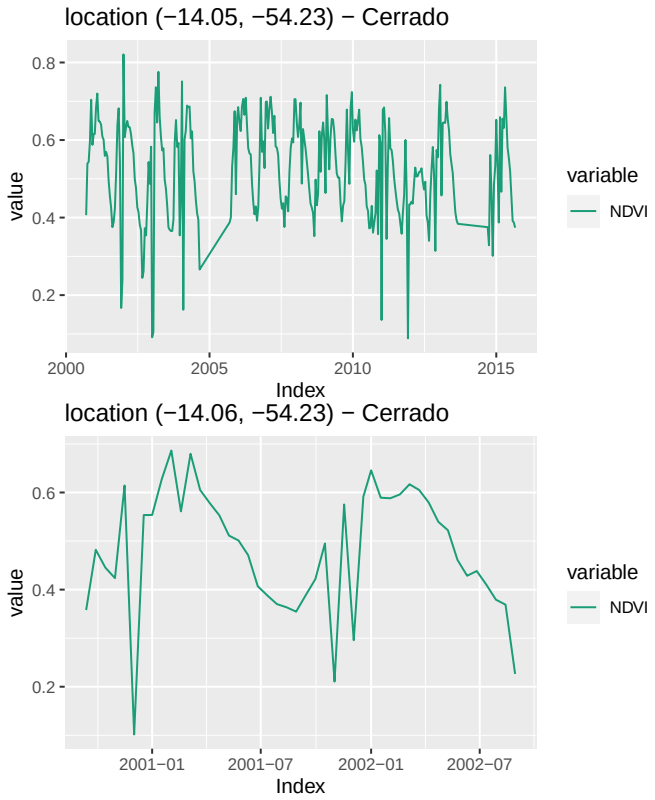


Figure 2.1: Plot of the first 'Cerrado' sample from data set

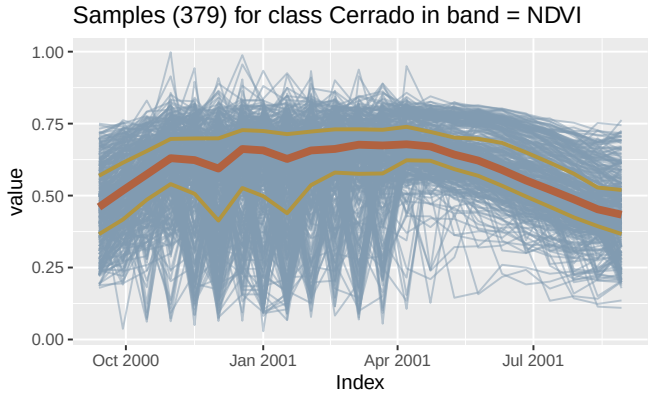


Figure 2.2: Plot of all Cerrado samples from data set

2.4 Obtaining time series data from data cubes

To get a time series in **sits**, one has to create a data cube first, as described above. 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()`.

The **sits** package enables the creation of a data cube based on files. In this case, these files should be organized as `raster stacks`. A raster stack is a single-layer raster object. Each file refers to a one-time instance and one spectral band. To allow users to create data cubes based on files, SITS needs to know the names of satellite and sensor and the directory names that contain the files.

```
# 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",
    satellite  = "TERRA",
    sensor     = "MODIS",
    name       = "Sinop",
    data_dir   = data_dir,
    parse_info = c("X1", "X2", "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.tb <- sits_get_data(cube      = raster_cube,
                           longitude = -55.57320,
                           latitude  = -11.50566,
                           bands     = c("NDVI", "EVI"))
plot(series.tb)
```

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 CSV files. Function `sits_get_data()` can get a CSV file path as an argument. The CSV file must provide, for each time series, its latitude and longitude, the start and end dates, 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 <- system.file("extdata/samples/samples_sinop_crop.csv",
```

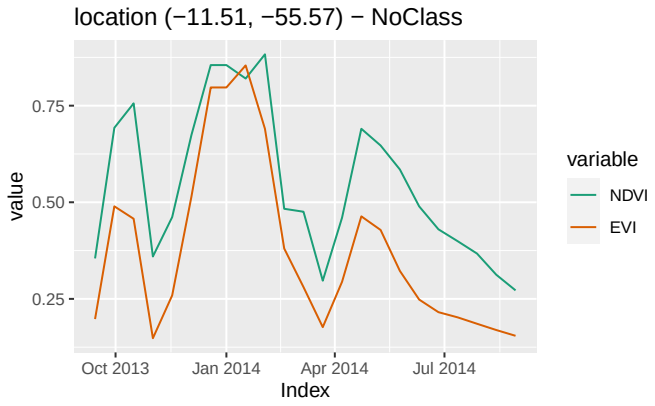


Figure 2.3: NDVI and EVI time series fetched from local raster cube.

```
                              package = "sits")
# get the points from a data cube in raster brick format
points <- sits_get_data(raster_cube, file = samples.csv)


# show the tibble with the points
points
```

```
#> # A tibble: 12 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]>
#>  4     -55.6    -11.8 2013-09-14 2014-08-29 Pasture  Sinop <tibble [23 x 3]>
#>  5     -55.7    -11.8 2013-09-14 2014-08-29 Forest   Sinop <tibble [23 x 3]>
#>  6     -55.6    -11.7 2013-09-14 2014-08-29 Forest   Sinop <tibble [23 x 3]>
#>  7     -55.7    -11.7 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
#>  8     -55.7    -11.7 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
#>  9     -55.7    -11.7 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
#> 10     -55.6    -11.8 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
#> 11     -55.6    -11.8 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
#> 12     -55.6    -11.8 2013-09-14 2014-08-29 Soy_Corn Sinop <tibble [23 x 3]>
```

A common situation is when users have samples available as shapefiles in point format. Since shapefiles contain only geometries, we need to provide information about the start and end times for which each label is valid. In this case, one should use the function `sits_get_data()` to retrieve data from a data cube based on the contents of the shapefile. The parameter `shp_attr` (optional) indicates the name of the column on the shapefile, which contains the label to

be associated with each time series; the parameter `.n_shp_pol` (defaults to 20) determines the number of samples to be extracted from each polygon.

```
# define the input shapefile (consisting of POLYGONS)
shp_file <- system.file("extdata/shapes/agriculture/parcel_agriculture.shp",
                        package = "sitsdata")

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

# define the name of attribute of the shapefile 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
data <- 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)
data
```

```
#> # A tibble: 10 x 7
#>    longitude latitude start_date end_date   label cube  time_series
#>        <dbl>    <dbl> <date>     <date>     <chr> <chr> <list>
#>  1     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  2     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  3     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  4     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  5     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  6     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  7     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  8     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#>  9     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
#> 10     -55.6    -11.8 2013-09-14 2014-08-29 <NA>  Sinop <tibble [23 x 3]>
```

Chapter 3

Satellite Image Time Series Filtering with sits

This chapter describes the time series filtering techniques available in the **sits** package. These include the Savitsky-Golay, Whittaker, and Kalman filters, as well as specialised filters for satellite images, which are the envelope filter for cloud removal.

3.1 Filtering techniques in sits

Satellite image time series generally is contaminated by atmospheric influence, geolocation error, and directional effects (Lambin and Linderman 2006). 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 (Atkinson et al. 2012). 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 `sits` have support for Savitzky–Golay (`sits_sgolay()`), Whitaker (`sits_whittaker()`), envelope (`sits_envelope()`) filters. The first two filters are commonly used in the literature, while the remaining two have been developed by the authors.

Various somewhat conflicting results have been expressed in relation to the time series filtering techniques for phenology applications. For example, in an investigation of phenological parameter estimation, Atkinson et al. (2012) found that the Whittaker and Fourier transform approaches were preferable to the double logistic and asymmetric Gaussian models. They applied the filters to preprocess MERIS NDVI time series for estimating phenological parameters in India. Comparing the same filters as in the previous work, Shao et al. (2016) found that only Fourier transform and Whittaker techniques improved interclass separability for crop classes and significantly improved overall classification accuracy. The authors used MODIS NDVI time series from the Great Lakes region in North America. Zhou et al. (2016) found that the asymmetric Gaussian model outperforms other filters over high latitude boreal biomes, while the Savitzky-Golay model gives the best reconstruction performance in tropical evergreen broadleaf forests. In the remaining biomes, Whittaker gives superior results. The authors compare all previously mentioned filters plus the Savitzky-Golay method for noise removal in MODIS NDVI data from sites spread worldwide in different climatological conditions. Many other techniques can be found in applications of satellite image time series such as curve fitting (Bradley et al. 2007), wavelet decomposition (Sakamoto et al. 2005), mean-value iteration, ARMD3-ARMA5, and 4253H (Hird and McDermid 2009). Therefore, any comparative analysis of smoothing algorithms depends on the adopted performance measurement.

One of the main uses of time series filtering is to reduce the noise and miss data produced by clouds in tropical areas. The following examples use data produced by the PRODES project (INPE 2017), which detects deforestation in the Brazilian Amazon rain forest through visual interpretation. This data set is called `samples_para_mixl8mod` and is provided together with the **sits** package. It has 617 samples from a region corresponding to the standard Landsat Path/Row 226/064. This is an area in the East of the Brazilian Pará state. It was chosen because of its huge cloud cover from November to March, which is a significant factor in degrading time series quality. Its NDVI and EVI time series were extracted from a combination of MOD13Q1 and Landsat8 images (to best visualize the effects of each filter, we selected only NDVI time series).

3.2 Common interface to sits filter functions

The **sits** package uses a common interface to all filter functions with the `sits_filter`. The function has two parameters: `data` for the dataset to be filtered and `filter` for the filter to be applied. To aid in data visualisation, all bands which are filtered have a suffix that is appended, as shown in the examples below. The following filters are available and described in what follows:

- Savitzky–Golay filter (`sits_sgolay`)
- Whittaker filter (`sits_whittaker`)
- Envelope filter (`sits_envelope`)

3.2.1 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 Madden (1978), 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 (Chen et al. 2004). 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 the original time series.

> The code below uses the `samples_para_mixl8mod` dataset provided by the sitsdata package.

```
data("samples_para_mixl8mod")

# Take NDVI band of the first sample data set
point.tb <- sits_select(samples_para_mixl8mod[1,], "ndvi")
# apply Savitzky-Golay filter
point_sg.tb <- sits_filter(point.tb, filter = sits_sgolay())
# plot the series
sits_merge(point_sg.tb, point.tb) %>% plot()
```

3.2.2 Whittaker filter

The Whittaker smoother attempts to fit a curve that represents the raw data, but is penalized if subsequent points vary too much (Atzberger and Eilers 2011). The Whittaker filter is a balancing between the residual to the original data and

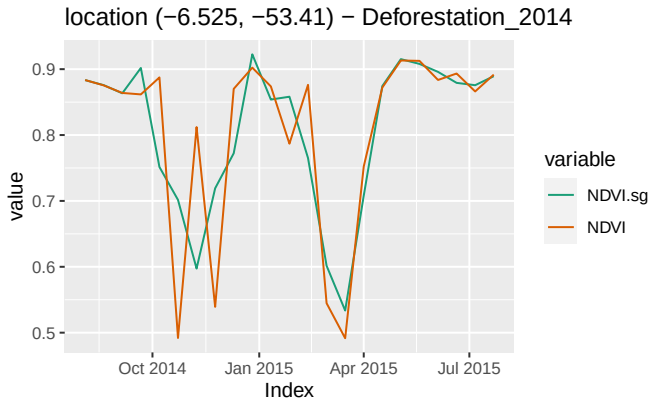


Figure 3.1: Savitzky-Golay filter applied on a one-year NDVI time series.

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 (Whittaker 1922), 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

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

is the third order difference matrix. The Whitakker filter can be a large but sparse optimization problem, as we can note from D matrix.

Whittaker smoother has been used only recently in satellite image time series investigations. According to Atzberger and Eilers (2011), the smoother has an advantage over other filtering techniques such as Fourier and wavelets as it does not assume signal periodicity. Moreover, the authors argue that it enables rapid processing of large amounts of data, and handles incomplete time series with missing values.

In the **sits** package, the Whittaker smoother has two parameters: `lambda` controls the degree of smoothing and `differences` the order of the finite difference penalty. The default values are `lambda = 1` and `differences = 3`. Users should be aware that increasing `lambda` results in much smoother data. When dealing with land use/land cover classes that include both natural vegetation and agriculture, a strong smoothing can reduce the amount of noise in natural vegetation (e.g., forest) time series; however, higher values of `lambda` reduce the information present in agricultural time series, since they reduce the peak values of crop plantations.

```
# filter NDVI band
point_ndvi <- sits_select(point_mt_6bands, bands = "NDVI")

# apply Whitaker filter to an NDVI time series from 2000 to 2016
# merge with the original data
# plot the original and the modified series
point_whit <- sits_filter(point_ndvi, sits_whittaker(lambda = 3))
point_whit %>%
  sits_merge(point_ndvi) %>%
  plot()
```

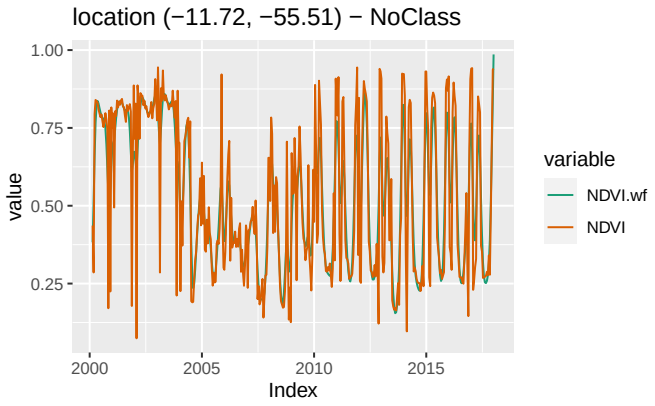


Figure 3.2: Whittaker smoother filter applied on 16-year NDVI time series

3.2.3 Envelope filter

This filter can generate a time series corresponding to the superior (inferior) bounding of the input signal. This is accomplished through a convoluting window (odd length) that attributes to the point i, in the resulting time series, the maximum (minimum) value of the points in the window. The i point corresponds to the central point of the window. It can be defined as

$$u_i = \max_k \left(\{x_k : |i - k| \leq 1\}\right),$$

whereas an lower dilation is obtained by

$$l_i = \min_k \left(\{x_k : |i - k| \leq 1\}\right).$$

Here, x is the input time series and, k and i are vector indices.

The `sits_envelope()` can combine both maximum and minimum windows sequentially. The function can receive a string sequence with `"U"` (for maximization) and `"L"` (for minimization) characters passed to its parameter. A repeated sequence of the same character is equivalent to one operation with a larger window. The sequential operations on the input time series produce the final filtered result that is returned.

The envelope filter can be viewed through mathematical morphology lenses, a very common field in digital image processing (Haralick, Sternberg, and Zhuang 1987). Here the operations of `"U"` and `"L"` correspond to the *dilation* and *erosion* morphological operators applied to univariate arrays (Vávra et al. 2004). Furthermore, the compound operation of *opening* and *closing* can be obtained by `"UL"` and `"LU"`, respectively. This technique has been applied to time series analysis in other fields (Accardo et al. 1997), but, to our knowledge, there is no application in satellite image time series literature.

In the following example, we can see an application of `sits_envelope()` function. There, we perform the *opening filtration* and *closing filtration* introduced by Vávra et al. (2004). The correspondent operations sequence is `"ULLULUUL"` and `"LUULULLU"`.

> The code below uses the `samples_para_mixl8mod` dataset provided by the sitsdata package.

```
# Take the NDVI band of the first sample data set
point.tb <- sits_select(samples_para_mixl8mod[1,], "ndvi")
# apply envelope filter (remove downward and upward noises)
point_env1.tb <- sits_filter(point.tb,
                             sits_envelope(operations = "ULLULUUL",
                                           bands_suffix = "OF"))
point_env2.tb <- sits_filter(point.tb,
                             sits_envelope(operations = "LUULULLU",
                                           bands_suffix = "CF"))
```

```
# plot the series
sits_merge(point_env1.tb, point_env2.tb) %>%
  sits_merge(point.tb) %>%
  plot()
```

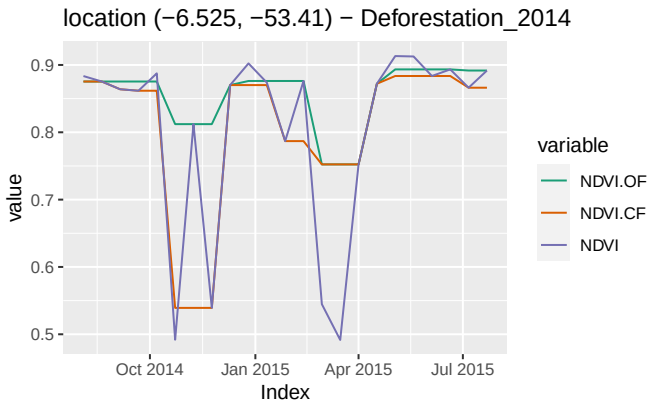


Figure 3.3: Envelope filter applied on one-year NDVI time series. The examples uses two morfological filters: opening filtration (.OF) and closing filtration (.CF).

Chapter 4

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. Representative samples lead to good classification maps. `sits` provides support for two clustering methods to test sample quality, which is agglomerative hierarchical clustering (AHC) and self-organizing maps (SOM).

4.1 Clustering for sample quality control

Recent results show that it is feasible to apply machine learning methods to SITS analysis in large areas of 100 million ha or more (Picoli et al. 2018; **???**; **???**; **???**). 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 (Maxwell, Warner, and Fang 2018); increasing the training sample size results in better classification accuracy (**???**). Therefore, using machine learning for SITS analysis requires large and good quality training sets.

One of the key challenges when using samples to train machine learning classifica-

tion models is assessing their quality. Noisy and imperfect training samples can have a negative effect on classification performance (**???**). There are two main sources of noise and errors in satellite image time series. One effect is *feature noise*, caused by clouds and inconsistencies in data calibration. The second effect is *class noise*, when the label assigned to the sample is wrongly attributed. Class noise effects are common on large data sets. In particular, interpreters tend to group samples with different properties in the same category. For this reason, one needs good methods for quality control of large training data sets associated with satellite image time series. Our work thus addresses the question: *How to reduce class noise in large training sets of satellite image time series?*

Many factors lead to *class noise* in SITS. One of the main problems is the inherent variability of class signatures in space and time. 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 idealised descriptions (e.g., "a savanna woodland has tree cover of 50% to 90% ranging from 8 to 15 meters in height"). However, in practice, the boundaries between classes are fuzzy and sometimes overlap, making it hard to distinguish between them. Class noise can also result from labeling errors. Even trained analysts can make errors in class attributions. Despite the fact that machine learning techniques are robust to errors and inconsistencies in the training data, quality control of training data can make a significant difference in the resulting maps.

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. `sits` provides support for two clustering methods to test sample quality: (a) Agglomerative Hierarchical Clustering (AHC); (b) Self-organizing Maps (SOM).

4.2 Hierachical clustering for Sample Quality Control

4.2.1 Creating a dendogram

Cluster analysis has been used for many purposes in satellite image time series literature ranging from unsupervised classification and pattern detection (Petitjean, Inglada, and Gançarskv 2011). Here, we are interested in the second use of clustering, using it as a way to improve training data to feed machine learning classification models. In this regard, cluster analysis can assist the identification of structural *time series patterns* and anomalous samples (**???**).

Agglomerative hierarchical clustering (AHC) is a family of methods that groups

elements using a distance function to associate a real value to a pair of elements. From this distance measure, we can compute 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. AHC approach is suitable for the purposes of samples data exploration due to its visualization power and ease of use (Keogh, Lin, and Truppel 2003). Moreover, AHC does not require a predefined number of clusters as an initial parameter. This is an important feature in satellite image time series clustering since defining the number of clusters present in a set of multi-attribute time series is not straightforward (Aghabozorgi, Shirkhorshidi, and Wah 2015).

The main result of the AHC method is a *dendrogram*. It is the ultrametric relation formed by the successive merges in the hierarchical process that can be represented by a tree. Dendrograms are quite useful to decide the number of clusters to partition the data. It shows the height where each merging happens, which corresponds to the minimum distance between two clusters defined by a *linkage criterion*. The most common linkage criteria are: *single-linkage*, *complete-linkage*, *average-linkage*, and *Ward-linkage*. 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* (Ward 1963). Ward's intuition is that clusters of multivariate observations, such as time series, should be approximately elliptical in shape (Hennig 2015). In `sits`, a dendrogram can be generated by `sits_dendrogram()`. The following codes illustrate how to create, visualize, and cut a dendrogram (for details, see `?sits_dendrogram()`).

4.2.2 Using a dendrogram to evaluate sample quality

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. We need to balance two objectives: get clusters as large as possible, and get clusters as homogeneous as possible with respect to their known classes. To help this process, `sits` provides `sits_dendro_bestcut()` function that computes an external validity index *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.

In this example, the height optimizes the ARI and generates 6 clusters. The ARI considers any pair of distinct samples and computes the following counts: (a) the number of distinct pairs whose samples have the same label and are in the same cluster; (b) the number of distinct pairs whose samples have the same label and are in different clusters; (c) the number of distinct pairs whose samples have different labels and are in the same cluster; and (d) the number of distinct pairs whose samples have the different labels and are in different clusters. Here, a and d consist in all agreements, and b and c all disagreements. The ARI is

obtained by:

$$ARI = \frac{a + d - E}{a + d + b + c - E},$$

where E is the expected agreement, a random chance correction calculated by

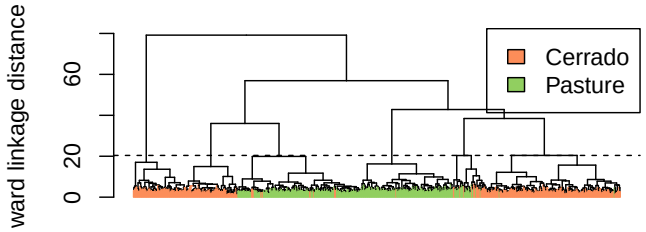
$$E = (a + b)(b + c) + (c + d)(b + d).$$

Unlike other validity indexes such as Jaccard ($J = a/(a + b + c)$), Fowlkes-Mallows ($FM = a/(a^2 + a(b + c) + bc)^{1/2}$), and Rand (the same as ARI without the E adjustment) indices, ARI is more appropriate either when the number of clusters is outweighed by the number of labels (and *vice versa*) or when the number of samples in labels and clusters are imbalanced (Hubert and Arabie 1985), which is usually the case.

```
# take a set of patterns for 2 classes
# create a dendrogram, plot, and get the optimal cluster based on ARI index
clusters <- sits::sits_cluster_dendro(cerrado_2classes,
                                      bands = c("ndvi", "evi"))

# show clusters samples frequency
sits::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
```



Note in this example that almost all 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 help to identify two kinds of confusion. The first is relative to those 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.

The result of the `sits_cluster` operation is a `sits_tibble` with one additional column, called "cluster". Thus, it is possible to remove clusters with mixed classes using standard `R` such as those in 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::sits_cluster_clean(clusters)
# show clusters samples frequency
sits_cluster_frequency(clean)
```

```
#>
#>             1   2   3   4   5   6 Total
#>   Cerrado 203   0   0  80   0  80   363
#>   Pasture   0 176  28   0 140   0   344
#>   Total   203 176  28  80 140  80   707
```

4.3 Using Self-organizing Maps for Sample Quality

4.3.1 Introduction to Self-organizing Maps

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 (**???**), where high-dimensional data is mapped into two dimensions, keeping the topological relations between data patterns. The input data is a set of training samples that are typical of a high dimension. For example, a time series of 25 instances of 4 spectral bands is a 100-dimensional data set. The general idea of SOM-based clustering is that, by projecting the high-dimensional data set of training samples into a 2D map, the units of the map (called "neurons") compete for each sample. It is expected that

good quality samples of each class should be close together in the resulting map. The neighbors of each neuron of a SOM map provide information on intra-class and inter-class variability.

The main steps of our proposed method for quality assessment of satellite image time series are shown in the figure below. The method uses self-organizing maps (SOM) to perform dimensionality reduction while preserving the topology of original datasets. Since SOM preserves the topological structure of neighborhoods in multiple dimensions, the resulting 2D map can be used as a set of clusters. Training samples that belong to the same class will usually be neighbors in 2D space. The neighbors of each neuron of a SOM map are also expected to be similar.

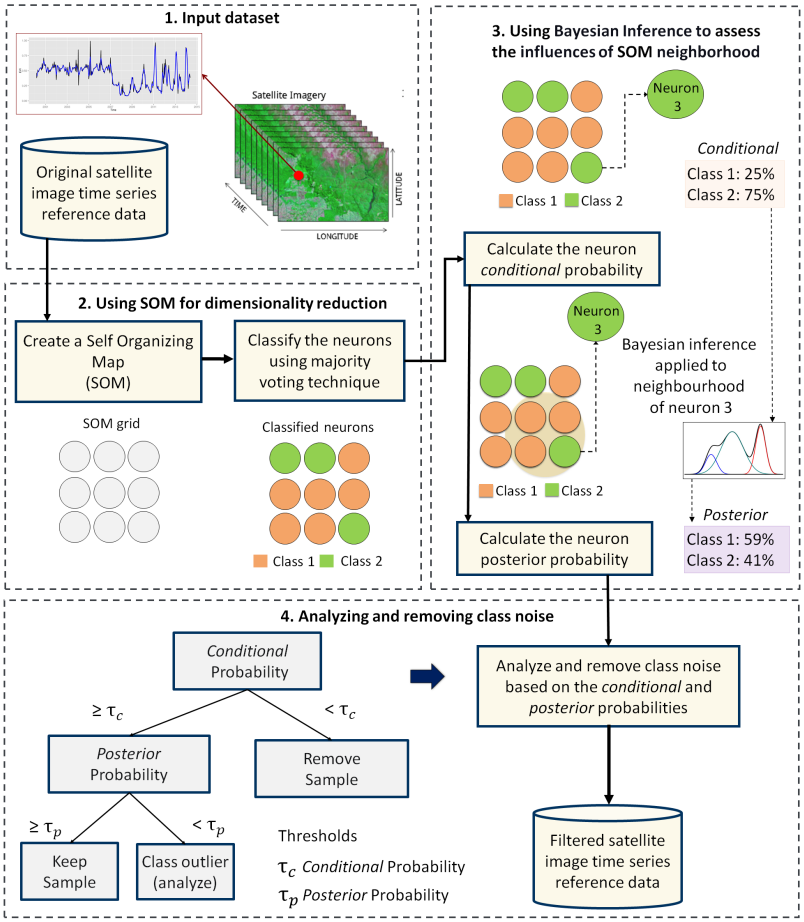


Figure 4.1: Using SOM for class noise reduction

As the figure shows, a SOM grid is composed of units called *neurons*. 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 weights of the BMU and its neighbors are updated so as to preserve their similarity [Kohonen2013]. This mapping and adjustment procedure is done in several iterations. At each step, the extent of the change in the neurons diminishes until a convergence threshold is reached. The result is a 2D mapping of the training set, where similar elements of the input are mapped to the same neuron or to nearby ones. The resulting SOM grid combines dimensionality reduction with topological preservation.

4.3.2 Using SOM for removing class noise

The process of clustering with SOM is done by `sits_som_map()`, which creates a self-organizing map and assesses the quality of the samples. The function has two parts. First, it computes a SOM grid, as discussed previously, where each sample is assigned to a neuron, and neurons are placed in the grid based on similarity. The second step is the quality assessment. Each neuron will be associated with a discrete probability distribution. Homogeneous neurons (those with a single class) 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.

Considering that each sample of the training set is assigned to a neuron, 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 in the same neuron. For example, if a neuron has 20 samples, of which 15 are labeled as "Pasture" and 5 as "Forest", all samples labeled "Forest" are assigned a prior probability of 25%. This is an indication that the "Forest" samples in this neuron 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 neighbours. Bayesian inference we estimate if these samples are noisy based on the samples of the neighboring neurons [Santos2021].

As an example of the use of SOM clustering for quality control of samples, we take a dataset containing a tibble with time series samples for 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 61,073 land use and cover samples divided into 14 classes: Natural Non-vegetated, Fallow-Cotton, Millet-Cotton, Soy-Corn, Soy-Cotton,

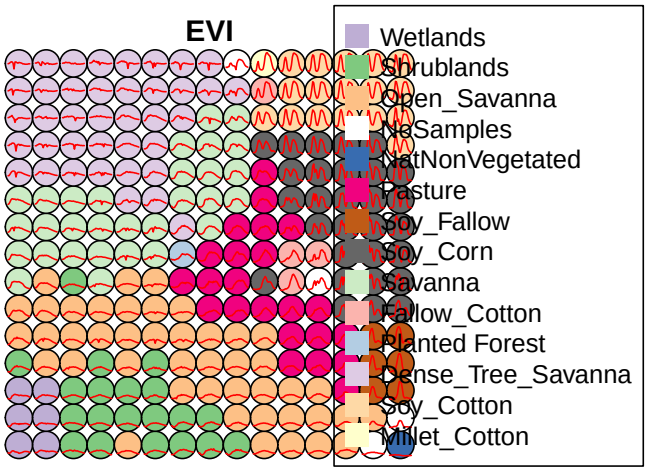
Soy-Fallow, Pasture, Shrublands (in Portuguese *Cerrado Rupestre*), Savanna (in Portuguese *Cerrado*, Dense Tree Savanna (in Portuguese *Cerradao*), Open Savanna (in Portuguese *Campo Cerrado*), Planted Forest, and (14) Wetlands. In the example below, we take only 10% of the samples for faster processing. Users are encouraged to run the example with the full set of samples.

```
# take only 10% of the samples
samples_cerrado_mod13q1_reduced <- sits_sample(samples_cerrado_mod13q1, frac = 0.1)
# clustering time series using SOM
som_cluster <-
    sits_som_map(
        samples_cerrado_mod13q1_reduced,
        grid_xdim = 15,
        grid_ydim = 15,
        alpha = 1.0,
        distance = "euclidean",
        rlen = 100
    )
```

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

```
plot(som_cluster)
```



Looking at the SOM grid, one can see that most of the neurons of a class are located close to each other. There are outliers, e.g., some "Open Savanna" neurons are located amidst "Shrublands" neurons. 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 shrublands is not always well defined; moreover, it is dependent on factors such as climate and latitude.

To identifies 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 uses the `prior_threshold` and `posterior_threshold` parameters according to the following rules:

- If the prior probability of a sample is less than `prior_threshold`, the sample is assumed to be noisy and tagged as "remove";
- If the prior probability is greater or equal to `prior_threshold` and the posterior probability 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   652
#> 2 clean    4416
```

4.3.3 Comparing Global Accuracy of Original and Clean Samples

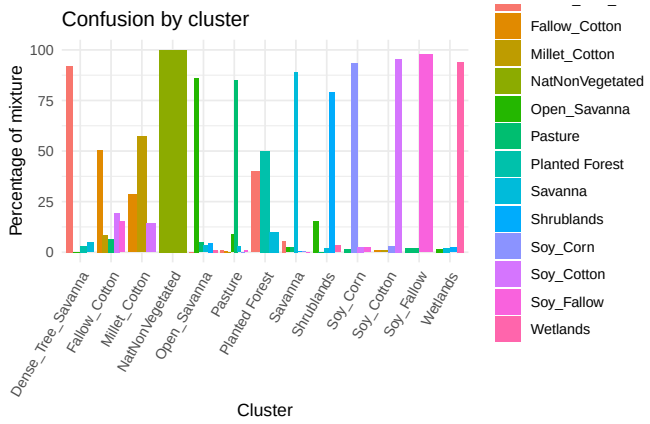
To compare the accuracy of the original and clean samples, we run a 5-fold validation on the original and on the cleaned sample. We use the function `sits_kfold_validate`. As the results show, the SOM procedure is useful, since the global accuracy improves from 91% to 95%.

```
assess_orig <- sits_kfold_validate(samples_cerrado_mod13q1_reduced,
                                   ml_method = sits_svm())
```

```
assess_new <- sits_kfold_validate(new_samples,
                                   ml_method = sits_svm())
```

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)
```



4.4 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.

Chapter 5

Machine Learning for Data Cubes using the SITS package

This chapter presents the machine learning 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.

5.1 Machine learning classification

`sits` has support for a variety of machine learning techniques: linear discriminant analysis, quadratic discriminant analysis, multinomial logistic regression, random forests, boosting, support vector machines, and deep learning. The deep learning methods include multi-layer perceptrons, 1D convolution neural networks and mixed approaches such as TempCNN (Pelletier, Webb, and Petitjean 2019) . In a recent review of machine learning methods to classify remote sensing data (Maxwell, Warner, and Fang 2018), the authors note that many factors influence the performance of these classifiers, including the size and quality of the training dataset, the dimension of the feature space, and the choice of the parameters. We support both *space-first, time-later* and *time-first, space-later* approaches. Therefore, the `sits` package provides functionality to explore the full depth of satellite image time series data.

When used in *time-first, space-later* approach, `sits` treats time series as a feature vector. To be consistent, the procedure aligns all time series from different years by its time proximity considering an given cropping schedule. Once aligned, the feature vector is formed by all pixel "bands". The idea is to have as many temporal attributes as possible, increasing the dimension of the classification space. In this scenario, statistical learning models are the natural candidates to deal with high-dimensional data: learning to distinguish all land cover and land use classes from trusted samples exemplars (the training data) to infer classes of a larger data set.

SITS provides support for the classification of both individual time series as well as data cubes. The following machine learning methods are available in SITS:

- Linear discriminant analysis (`sits_lda`)
- Quadratic discriminant analysis (`sits_qda`)
- Multinomial logit and its variants 'lasso' and 'ridge' (`sits_mlr`)
- Support vector machines (`sits_svm`)
- Random forests (`sits_rfor`)
- Extreme gradient boosting (`sits_xgboost`)
- Deep learning (DL) using multi-layer perceptrons (`sits_deeplearning`)
- DL with 1D convolutional neural networks (`sits_FCN`)
- DL using 1D version of ResNet (`sits_ResNet`)
- DL combining 1D CNN and multi-layer perceptron networks (`sits_TempCNN`)
- DL using a combination of long-short term memory (LSTM) and 1D CNN (`sits_LSTM-FCN`)

5.2 Data used in the machine learning examples

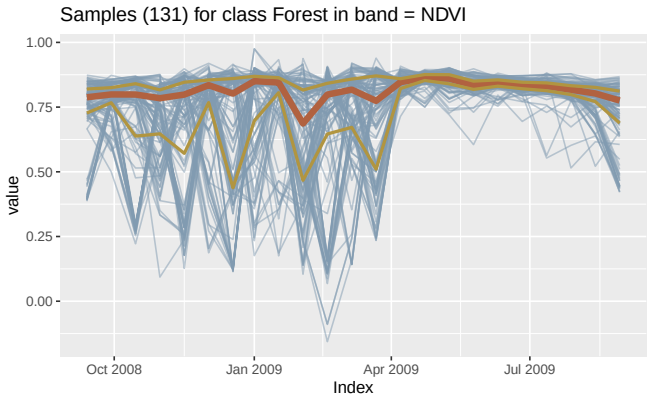
For the machine learning examples, we use a data set containing a sits tibble with time series samples from Brazilian Mato Grosso State (Amazon and Cerrado biomes). The samples are from many sources. It has 9 classes ("Cerrado", "Fallow_Cotton", "Forest", "Millet_Cotton", "Pasture", "Soy_Corn", "Soy_Cotton", "Soy_Fallow", "Soy_Millet"). Each time series comprehends 12 months (23 data points) from MOD13Q1 product, and has 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" (Picoli et al. 2018), and is available in the R package "sitsdata", which is downloadable from the website associated to the "e-sensing" project. The examples below use two out of six bands ("ndvi", "evi") for training and classification. In practice, we suggest that users include additionally at least the "nir" and "mir" bands.

5.3 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. When applied to a large data sample, the result is the set of all samples for each label and each band, as shown in the example below, where we plot the raw distribution of the samples with "Forest" label in the "ndvi" band.

```
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()
```



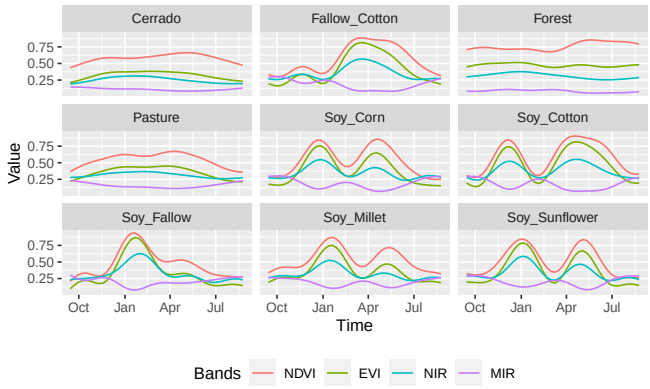
In the above plot, the thick red line is the median value for each time instance and the yellow lines are the first and third interquartile ranges. Visually, one can see that samples labelled as "Forest" are distinguishable from those of "Cerrado" and "Pasture"; in turn, these latter classes have many similar features and required sophisticated methods for distinction.

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`(Maus et al. 2019), which implements the TWDTW time series matching method described in Maus et al. (2016). 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.

5.4 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 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 the `sits_classify` function. In the examples that follow, we show how to apply each method for the classification of a single time series. Then, we disscuss how to classify full data cubes.

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 following examples illustrate how to train a dataset and classify an individual time series using the different machine learning techniques. First we use the `sits_train` function with two parameters: the training dataset

(described above) and the chosen machine learning model (in this case, a random forest classifier). The trained model is then used to classify a time series from Mato Grosso Brazilian state, using `sits_classify`. The results can be shown in text format using the function `sits_show_prediction` or graphically using `plot`.

5.5 Random forests

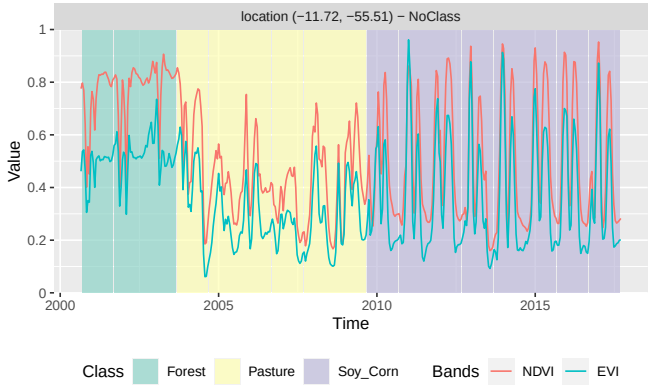
The Random forest uses the idea of *decision trees* as its base model. It combines many decision trees via *bootstrap* procedure and *stochastic feature selection*, developing a population of somewhat uncorrelated base models. The final classification model is obtained by a majority voting schema. This procedure decreases the classification variance, improving prediction of individual decision trees.

Random forest training process is essentially nondeterministic. It starts by growing trees through repeatedly random sampling-with-replacement the observations set. At each growing tree, the random forest considers only a fraction of the original attributes to decide where to split a node, according to a *purity criterion*. This criterion is used to identify relevant features and to perform variable selection. This decreases the correlation among trees and improves the prediction performance. Two often-used impurity criteria are the *Gini* index and the *permutation* measure. The Gini index considers the contribution of each variable which improves the spliting criteria for building tress. Permutation increases the importance of variables that have a positive effect on the prediction accuracy. The splitting process continues until the tree reaches some given minimum nodes size or a minimum impurity index value.

One of the advantages of the random forest model is that the classification performance is mostly dependent on the number of decision trees to grow and of the "importance" parameter, which controls the purity variable importance measures. SITS provides a `sits_rfor` function which is a front-end to the `ranger` package(Wright and Ziegler 2017); its two main parameters are: `num_trees` (number of trees to grow) and `importance`, the variable importance criterion. Possible values for `importance` are: `none`, `impurity` (Gini index), and `permutation`, the default being `impurity`.

```
# Retrieve the set of samples (provided by EMBRAPA) from the
# Mato Grosso region for train the Random Forest model.
rfor_model <- sits_train(samples_matogrosso_mod13q1, sits_rfor())
# Classify using Random Forest 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 = 0.2, bands_suffix = "") %>%
    sits_classify(rfor_model)
# plot classification
```

```
class.tb %>%
    plot(bands = c("NDVI", "EVI"))
```




```
# show the results of the prediction
sits_show_prediction(class.tb)
```

```
#> # 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 the tendency of the random forest classifier to be robust to outliers and to be able to deal with irrelevant inputs (Hastie, Tibshirani, and J. 2009). Performs internal variable selection helps the results be robust to outliers and noise, a common feature in image time series. However, despite being robust, random forest tend to overemphasize some variables and thus rarely turn out to be the classifier with the smallest error. One reason is that the performance

of random forest tends to stabilise after a part of the trees are grown (Hastie, Tibshirani, and J. 2009). Random forest classifiers can be quite useful to provide a baseline to compare with more sophisticated methods.

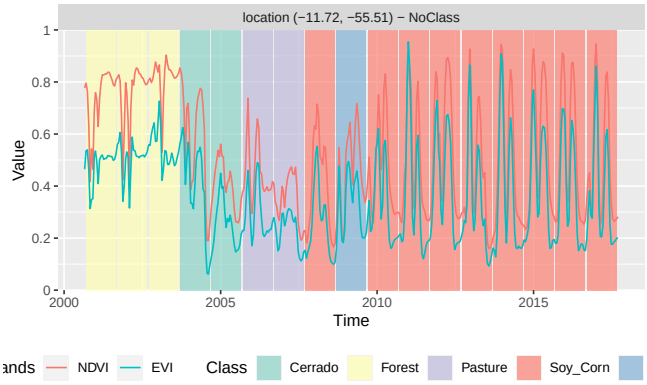
5.6 Support Vector Machines

Given a multidimensional data set, the Support Vector Machine (SVM) method finds an optimal separation hyperplane that minimizes misclassifications (Cortes and Vapnik 1995). Hyperplanes are linear $(p-1)$-dimensional boundaries and define linear partitions in the feature space. The solution for the hyperplane coefficients depends only on those samples that violates the maximum margin criteria, the so-called *support vectors*. All other points far away from the hyperplane does not exert any influence on the hyperplane coefficients which let SVM less sensitive to outliers.

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. In this manner, 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. The use of kernels are an efficient computational strategy to produce nonlinear boundaries in the input attribute space an hence can 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 (Mountrakis, Im, and Ogole 2011).

In `sits`, SVM is implemented as a wrapper of `e1071` R package that uses the `LIBSVM` implementation (Chang and Lin 2011), `sits` 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.

```
# 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(kernel = "radial",
                                             cost = 10))
# Classify using SVM model and plot the result
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.25, bands_suffix = "") %>%
    sits_classify(svm_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



The result is mostly consistent of what one could expect by visualising the time series. 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 2008 onwards. However, the result shows some inconsistencies. First, since the training dataset does not contain a 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 and grasslands are mixed. This misinterpretation needs to be corrected in post-processing by applying a time-dependent rule (see the main SITS vignette and the post-processing methods vignette). Also, the classification for year 2009 is "Soy-Millet", which is different from the "Soy-Corn" label assigned from the other years from 2008 to 2017. To test if this result is inconsistent, one could apply spatial post-processing techniques, as discussed in the main SITS vignette and in the post-processing one.

One of the drawbacks of using the `sits_svm` method is its sensitivity to its parameters. Using a linear or a polynomial kernel fails to produce good results. If one varies the parameter `cost` (cost of contraints violation) from 100 to 1, the results can be strinkgly different. Such sensitity to the input parameters points to a limitation when using the SVM method for classifying time series.

5.7 Extreme Gradient Boosting

Boosting techniques are based on the idea of starting from a weak predictor and then improving performance sequentially by fitting better model at each iteration. It starts by fitting a simple classifier to the training data. Then it uses the residuals of the regression to build a better prediction. Typically, the base classifier is a regression tree. Although both random forests and boosting use trees for classification, there is an important difference. In the random forest classifier, the same random logic for tree selections is applied at every step (Efron and Hastie 2016). Boosting trees are built to improve on previous result, by applying finer divisions that improve the performance. The performance of random forests generally increases with the number of trees until it becomes stable; however, the number of trees grown by boosting techniques cannot be

too large, at the risk of overfitting the model.

Gradient boosting is a variant of boosting methods where the cost function is minimized by agradient descent algorithm. Extreme gradient boosting (Chen and Guestrin 2016), called "XGBoost", improves by using an efficient approximation to the gradient loss function. The algorithm is fast and accurate. XGBoost is considered one of the best statistical learning algorithms available and has won many competitions; it is generally considered to be better than SVM and random forests. However, actual performance is controlled by the quality of the training dataset.

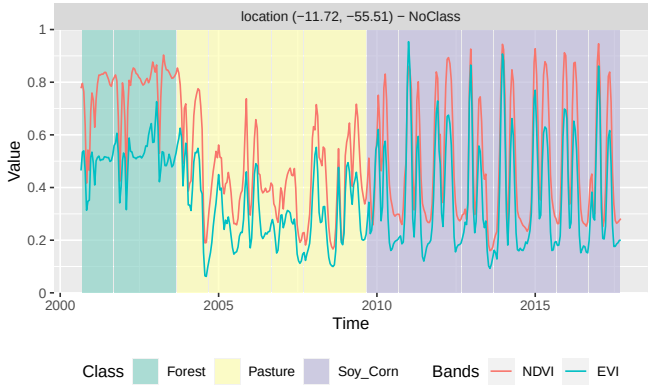
In SITS, the XGBoost method is implemented by the `sits_xbgoost()` function, which is based on "XGBoost" R package and has five parameters that require tuning. The learning rate `eta` varies from 0 to 1, but show be kept small (default is 0.3) to avoid overfitting. The minimim loss value `gamma` specifies the minimum reduction required to make a split. Its default is 0, but increasing it makes the algorithm more conservative. The maximum depth of a tree `max_depth` controls how deep tress are to be built. In principle, it should not be largem since higher depth trees lead to overfitting (default is 6.0). 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` parameters controls the maximum number of boosting interactions; its default is 100, which has proven to be sufficient in the SITS. 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_matogrosso_mod13q1, sits_xgboost())
```

```
#> [1]  train-mlogloss:2.004707+0.001064    test-mlogloss:2.017435+0.001713
#> Multiple eval metrics are present. Will use test_mlogloss for early stopping.
#> Will train until test_mlogloss hasn't improved in 20 rounds.
#>
#> [11] train-mlogloss:0.579262+0.002762    test-mlogloss:0.703353+0.016109
#> [21] train-mlogloss:0.171149+0.001497    test-mlogloss:0.310619+0.018772
#> [31] train-mlogloss:0.070559+0.000833    test-mlogloss:0.201973+0.016472
#> [41] train-mlogloss:0.041928+0.000694    test-mlogloss:0.165725+0.014463
#> [51] train-mlogloss:0.033341+0.000360    test-mlogloss:0.154319+0.013683
#> [61] train-mlogloss:0.031651+0.000247    test-mlogloss:0.151723+0.013116
#> [71] train-mlogloss:0.031114+0.000358    test-mlogloss:0.150990+0.012583
#> [81] train-mlogloss:0.030521+0.000382    test-mlogloss:0.150283+0.012527
#> [91] train-mlogloss:0.030141+0.000562    test-mlogloss:0.149470+0.012279
#> [100]    train-mlogloss:0.029954+0.000459    test-mlogloss:0.148981+0.012033
```

```
# Classify using SVM model and plot the result
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.25, bands_suffix = "") %>%
```

```
    sits_classify(xgb_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



In general, the results from the extreme gradient boosting model are similar to the Random Forest model. However, for each specific study, users need to perform validation. See the function `sits_kfold_validate` for more details.

5.8 Deep learning using multi-layer perceptrons

Using the `keras` package (Chollet and Allaire 2018) as a backend, SITS supports the following =deep learning techniques, as described in this section and the next ones. The first method is that of feedforward neural networks, or multi-layer perceptron (MLPs). These are the quintessential deep learning models. The goal of a multilayer perceptrons is to approximate some function f. For example, for a classifier $y = f(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself (Goodfellow, Bengio, and Courville 2016).

Specifying a MLP requires some work on customization, which requires some amount of trial-and-error by the user, since there is no proven model for classification of satellite image time series. The most important decision is the number of layers in the model. Initial tests indicate that 3 to 5 layers are enough to produce good results. 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.

Three other important parameters for an MLP are: (a) the activation function; (b) the optimization method; (c) the dropout rate. The activation function the activation function of a node defines the output of that node given an input or set of inputs. Following standard practices (Goodfellow, Bengio, and Courville 2016), we recommend the use of the "relu" and "elu" functions. The optimization method is a crucial choice, and the most common choices are gradient descent algorithm. These methods aim to maximize an objective function by updating the parameters in the opposite direction of the gradient of the objective function (Ruder 2016). 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.

The dropout rates have a huge impact on the performance of MLP classifiers. Dropout is a technique for randomly dropping units from the neural network during training (Srivastava et al. 2014). By randomly discarding some neurons, dropout reduces overfitting. It is a counter-intuitive idea that works well. Since the purpose of a cascade of neural nets is to improve learning as more data is acquired, discarding some of these neurons may seem a waste of resources. In fact, as experience has shown (Goodfellow, Bengio, and Courville 2016), this procedures prevents an early convergence of the optimization to a local minimum. Thus, in practice, dropout rates between 50% and 20% are recommended for each layer.

In the following example, we classify the same data set using an example of the `deep learning` method. The parameters for the MLP are: (a) Three layers with 512 neurons each, specified by the parameter `layers`; (b) Using the 'elu' activation function; (c) dropout rates of 50%, 40% and 30% for the layers; (d) the "optimizer_adam" as optimizer (default value); (e) a number of training steps (`epochs`) of 75; (f) a `batch_size` of 128, which indicates how many time series are used for input at a given steps; (g) a validation percentage of 20%, which means 20% of the samples will be randomly set side for validation. In practice, users may want to increase the number of epochs and the number of layers. 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. If a better performance is required, users should try to use the convolutional models descibed below. To simplify the vignette generation, the `verbose` option has been turned off. The default value is on. After the model has been generated, we plot its training history.

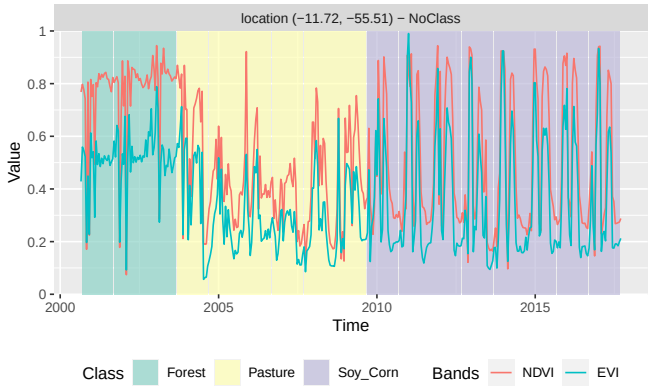
```
# train a machine learning model for the Mato Grosso data using an MLP
mlp_model <- sits_train(samples_matogrosso_mod13q1,
                        sits_deeplearning(
                        layers           = c(128, 128, 128),
                        activation       = "elu",
```

```
                        dropout_rates    = c(0.50, 0.40, 0.30),
                        epochs           = 80,
                        batch_size       = 128,
                        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_4bands <- sits_select(point_mt_6bands,
                               bands = c("NDVI", "EVI", "NIR", "MIR"))
class.tb <- point_mt_4bands %>%
    sits_classify(mlp_model) %>%
    plot(bands = c("ndvi", "evi"))
```



5.9 1D Convolutional Neural Networks

Convolutional neural networks (CNN) are a variety of deep learning methods where a convolution filter (sliding window) is applied to the input data. In the case of time series, a 1D CNN works by applying a moving window to the series. Using convolution filters is a way to incorporate temporal autocorrelation information in the classification. The result of the convolution is another time series. Rußwurm and Korner (2017) 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. Also, 1D-CNNs with a suitable convolution window make the classifier more robust to moderate noise, e.g. intermittent presence of clouds.

SITS includes four different variations of 1D-CNN, described in what follows. The first one is a "full Convolutional Neural Network"(Wang, Yan, and Oates

2017), implemented in the `sits_FCN` function. FullCNNs are cascading networks, where the size of the input data is kept constant during the convolution. After the nvolutions have been applied, the model includes a global average pooling layer which reduces the number of parameters, and highlights which parts of the input time series contribute the most to the classification (Fawaz et al. 2019). The fullCNN architecture proposed in Wang, Yan, and Oates (2017) has three convolutional layers. Each layer performs a convolution in its input and uses batch normalization for avoiding premature convergence to a local minimum. Batch normalisation is an alternative to dropout (Ioffe and Szegedy 2015). The result is to a ReLU activation function. The result of the third convolutional block is averaged over the whole time dimension which corresponds to a global average pooling layer. Finally, a traditional softmax classifier is used to get the classification results (see figure below).

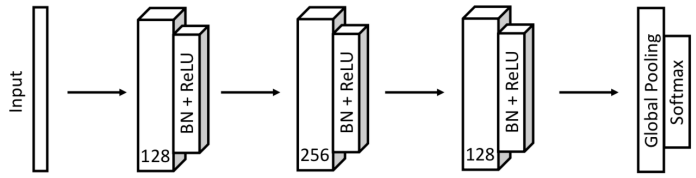


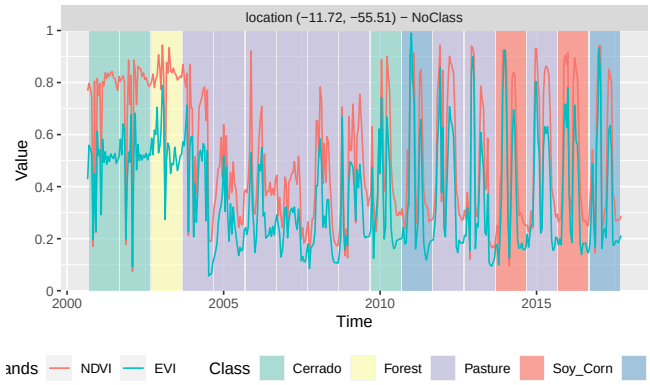
Figure 5.1: Structure of fullCNN architecture (source: Wang et al.(2017))

The `sits_FCN` function uses the architecture proposed by Wang as its default, and allows the users to experiment with different settings. The `layers` parameter controls the number of layers and the number of filters in each layer. The `kernels` parameters controls the size of the convolution kernels for earh layer. In the example below, the first convolution uses 64 filters with a kernel size of 8, followed by a second convolution of 128 filters with a kernel size of 5, and a third and final convolutional layer with 64 filters, each one with a kernel size to 3.

```
# train a machine learning model using deep learning
fcn_model <- sits_train(samples_matogrosso_mod13q1,
                        sits_FCN(
                            layers           = c(64, 256, 64),
                            kernels          = c(8, 5, 3),
                            activation       = 'relu',
                            L2_rate          = 1e-06,
                            epochs           = 100,
                            batch_size       = 128,
                            verbose          = 0) )
# show training evolution
plot(fcn_model)
```

Then, we classify a 16-year time series using the FCN 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_classify(fcn_model) %>%
    plot(bands = c("ndvi", "evi"))
```



5.10 Residual 1D CNN Networks (ResNet)

The Residual Network (ResNet) is a variation of the fullCNN network proposed by Wang, Yan, and Oates (2017). ResNet is composed of 11 layers (see figure below). ResNet is a deep network, by default divided in three blocks of three 1D CNN layers each. Each block corresponds to a fullCNN network archicture. The output of each block is combined with a shortcut that links its output to its input. The idea is avoid the so-called "vanishing gradient", which occurs when a deep learning network is trained based gradient optimization methods(Hochreiter 1998). As the networks get deeper, otimising them becomes more difficult. Including the input layer of the block at its end is a heuristic that has shown to be effective. In a recent review of time series classification methods using deep learning, Fawaz et al. state the RestNet and fullCNN have the best performance on the UCR/UEA time series test archive (Fawaz et al. 2019).

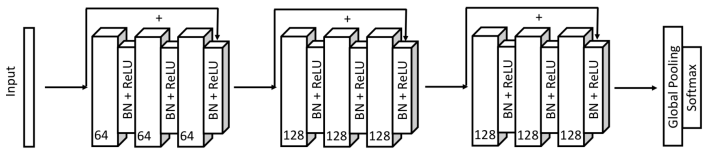


Figure 5.2: Structure of ResNet architecture (source: Wang et al.(2017))

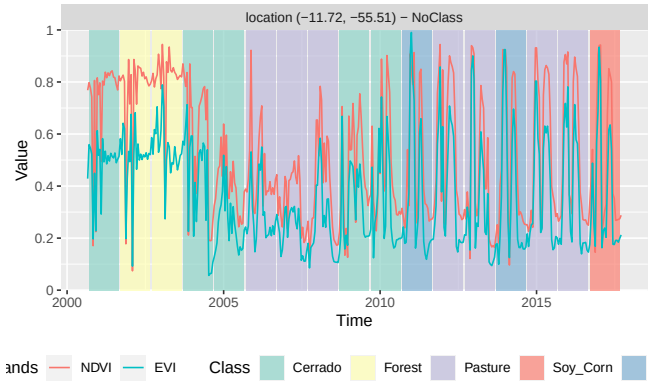
In SITS, the ResNet is implemented using the `sits_resnet` function. The default parameters are those proposed by Wang, Yan, and Oates (2017), and

we also benefited from the code provided by Fawaz et al. (2019) (https://github.com/hfawaz/dl-4-tsc). 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 Wang, Yan, and Oates (2017) and Fawaz et al. (2019). In the case of the 2-band Mato Grosso data set, the estimated accuracy is 95.8%.

```
# train a machine learning model using ResNet
point_mt_4bands <- sits_select(point_mt_6bands,
                               bands = c("NDVI", "EVI", "NIR", "MIR"))
resnet_model <- sits_train(samples_matogrosso_mod13q1,
                       sits_ResNet(
                          blocks               = c(32, 64, 64),
                          kernels              = c(9, 7, 3),
                          activation           = 'relu',
                          epochs               = 100,
                          batch_size           = 128,
                          validation_split     = 0.2,
                          verbose              = 0) )
# show training evolution
plot(resnet_model)
```

Then, we classify a 16-year time series using the DL 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_classify(resnet_model) %>%
    plot(bands = c("NDVI", "EVI"))
```



5.11 Combined 1D CNN and multi-layer perceptron networks

The combination of 1D CNNs and multi-layer perceptron models for satellite image time series classification was first proposed in Pelletier, Webb, and Petitjean (2019). The so-called "tempCNN" architecture consists of a number of 1D-CNN layers, similar to the fullCNN model discussed above, whose output is fed into a set of multi-layer perceptrons. The original tempCNN architecture is composed of 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 gradiente effect, including dropout, regularization, and batch normalisation. In the tempCNN paper (Pelletier, Webb, and Petitjean 2019), the authors compare favourably the tempCNN model with the Recurrent Neural Network proposed by Rußwurm and Körner (2018) for land use classification. The figure below shows the architecture of the tempCNN model.

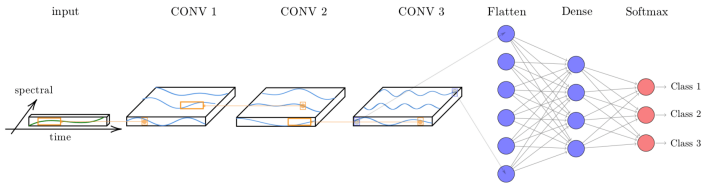


Figure 5.3: Structure of tempCNN architecture (source: Pelletier et al.(2019))

The function `sits_tempCNN` implements the model, using the default parameters proposed by Pelletier, Webb, and Petitjean (2019). The code has been derived from the Python source provided by the authors (https://github.com/charlotte-pel/temporalCNN). The parameter `cnn_layers` controls the number of 1D-CNN layers and the size of the filters applied at each layer; the parameter

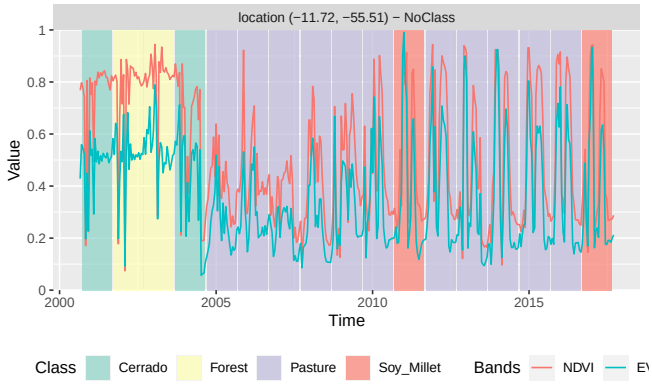
`cnn_kernels` indicates the size of the convolution kernels. Activation, regularisation for all 1D-CNN layers are set, respectively, by the `cnn_activation`, `cnn_L2_rate`. The dropout rates for each 1D-CNN layer are controlled individually by the parameter `cnn_dropout_rates`. The parameters `mlp_layers` and `mlp_dropout_rates` allow the user to set the number and size of the desired MLP layers, as well as their dropout_rates. The activation of the MLP layers is controlled by `mlp_activation`. By default, the function uses the ADAM optimizer, but any of the optimizers available in the `keras` package can be used. 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. In the case of the 2-band Mato Grosso data set, the estimated accuracy is 95.5%.

```
# train a machine learning model using tempCNN
tCNN_model <- sits_train(samples_matogrosso_mod13q1,
                         sits_TempCNN(
                            cnn_layers           = c(32, 32, 32),
                            cnn_kernels          = c(5, 5, 5),
                            cnn_activation       = 'relu',
                            cnn_L2_rate          = 1e-06,
                            cnn_dropout_rates    = c(0.50, 0.50, 0.50),
                            mlp_layers           = c(256),
                            mlp_activation       = 'relu',
                            mlp_dropout_rates    = c(0.50),
                            epochs               = 60,
                            batch_size           = 128,
                            validation_split     = 0.2,
                            verbose              = 0) )

# show training evolution
plot(tCNN_model)
```

Then, we classify a 16-year time series using the DL 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_classify(tCNN_model) %>%
    plot(bands = c("ndvi", "evi"))
```



5.12 LSTM Convolutional Networks for Time Series Classification

Given the success of 1D-CNN networks for time series classification, there have been a number of variants proposed in the literature. One of these variants is the LSTM-CNN network (Karim et al. 2018), where a fullCNN is combined with long short term memory (LSTM) recurrent neural network. LSTMs are an improved version of recurrent neural networks (RNN). An RNN is a neural network that includes a state vector, which is updated every time step. In this way, a RNN combines an input vector with information that is kept from all previous inputs. One can conceive of RNN as networks that have loops, allowing information to be passed from one step to to the next. In theory, a RNN would be able to handle long-term dependencies between elements of the input vectors. In practice, they are prone to the exhibit the "vanishing gradient" effect. As discussed above, in deep neural networks architectures with gradient descent optimization the gradient function can approach zero, thus impeding training to be done efficiently. LSTM improve on RNN architecture by including the additional feature of being able to regulate whether or not new information should be included on the cell state. LSTM unit also include a forget gate, which is able to discard the previous information stored in the cell state. Thus, a LSTM unit is able to remember values over arbitrary time intervals.

Karim et al. (2019) consider that LSTM networks are "well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series". The authors proposed a mixed LSTM-CNN architecture, composed of two paralel data streams: a 3-step CNN such as the one implemented in `sits_FCN` (see above) combined with a data stream consisting of an LSTM unit, as shown in the figure below. In Karim et al. (2018), the authors argue the LSTM-CNN model is capable of a better performance in the UCR/UEA time series test set than architectures such as ResNet and fullCNN.

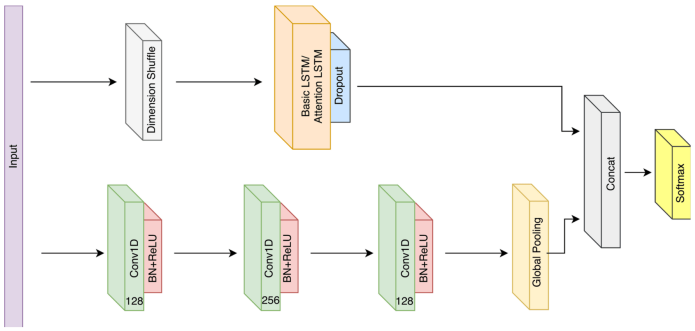


Figure 5.4: LSTM Fully Convolutional Networks for Time Series Classification (source: Karim et al.(2019))

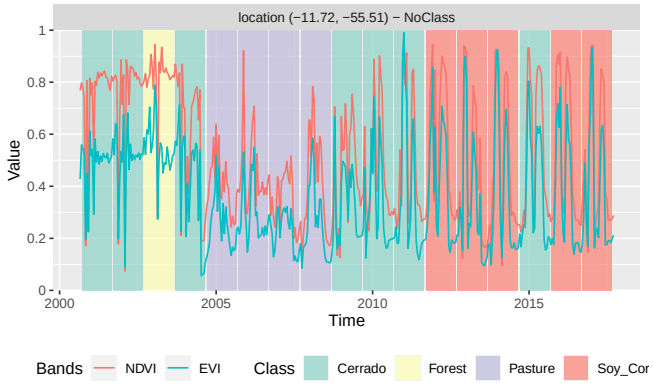
In the SITS package, the combined LSTM-CNN architecture is implemented by the `sits_LSTM_CNN` function. The default values are similar those proposed by Karim et al. (2019). The parameter `lstm_units` controls the number of units in the LSTM cell at every time step of the network. Karim et al. (2019) proposes an LSTM with 8 units, each with a dropout rate of 80%, which are controlled by parameters `lstm_units` and `lstm_dropout`. In initial experiments, we got a better performance with an LSTM with 16 units. As proposed by Karim et al. (2019), the CNN layers have filter sizes of {128, 256, 128} and kernel convolution sizes of {8, 5, 3}, controlled by the parameters `cnn_layers` and `cnn_kernels`. One should experiment with these parameters, and consider the simulations carried out by Pelletier, Webb, and Petitjean (2019) (see above), where the authors found that an FCN network of sizes {64, 64, 64} with kernels sizes of {5, 5, 5} had best performance in their case study. In this example, the estimated accuracy of the model was 94.7%.

```
lstm_fcn_model <- sits_train(samples_matogrosso_mod13q1,
                         sits_LSTM_FCN(
                            lstm_units          = 16,
                            lstm_dropout        = 0.80,
                            cnn_layers          = c(64, 64, 64),
                            cnn_kernels         = c(8, 5, 3),
                            activation          = 'relu',
                            epochs              = 120,
                            batch_size          = 128,
                            validation_split    = 0.2,
                            verbose             = 0) )

# show training evolution
plot(lstm_fcn_model)
```

Then, we classify a 16-year time series using the DL 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_classify(lstm_fcn_model) %>%
    plot(bands = c("ndvi", "evi"))
```



Chapter 6

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.

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

In broad terms, the cloud computing model is one where large satellite-generated data sets are archived on cloud services, which also provide computing facilities to process them. By using cloud services, users can share big Earth observation databases and minimize the amount of data download. Investment in infrastructure is minimised and sharing of data and software increases. However, data available in the cloud is best organised for analysis by creating data cubes.

Generalising Appel and Pebesma (2019), we consider that a data cube is a four-dimensional structure with dimensions x (longitude or easting), y (latitude or northing), time, and bands. Its spatial dimensions refer to a single spatial reference system (SRS). Cells of a data cube have a constant spatial size (with regard to the cube's SRS). The temporal dimension is specified by a set of intervals. For every combination of dimensions, a cell has a single value. Data cubes are particularly amenable for machine learning techniques; their data cane be transformed into arrays in memory, which can be fed to training and

classification algorithms. Given the widespread availability of large data sets of Earth observation data, there is a growing interest in organising large sets of data into "data cubes".

As explained below, a data cube is the data type used in `sits` to handle dense raster data. Many of the operations involve creating, transforming and analysing data cubes.

6.2 Defining a data cube using files organised as raster bricks

The SITS package enables uses to create data cube based on files. In this case, these files should be organized as `raster bricks`. A RasterBrick is a multi-layer raster object used by the *R* `raster` package. Each brick is a multi-layer file, containing different time instances of one spectral band. To allow users to create data cubes based on files, SITS needs to know what is the timeline of the data sets and what are the names of the files that contain the RasterBricks. The example below shows one bricks containing 392 time instances of the "ndvi" band for the years 2000 to 2016. The timeline is available as part of the SITS package. In this example, as in most cases using raster bricks, images are stored as GeoTiff files.

Since GeoTiff files do not contain information about satellites and sensors, it is best practice to provide information on satellite and sensor.

```
# 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",
                         satellite = "TERRA",
                         sensor  = "MODIS",
                         name = "Sinop",
                         data_dir = data_dir,
                         parse_info = c("X1", "X2", "band", "date"),
)

# get information on the data cube
raster_cube %>% dplyr::select(source, satellite, sensor)
```

```
#> # A tibble: 1 x 3
#>   source satellite sensor
#>   <chr>  <chr>     <chr>
#> 1 LOCAL  TERRA     MODIS
```

```
# get information on the coverage
raster_cube %>% dplyr::select(xmin, xmax, ymin, ymax)
```

```
#> # A tibble: 1 x 4
#>        xmin      xmax      ymin      ymax
#>       <dbl>     <dbl>     <dbl>     <dbl>
#> 1 -6087719. -5984864. -1355172. -1256255.
```

To create the raster cube, we a set of consistent raster bricks (one for each satellite band) and a `timeline` that matches the input images of the raster brick. Once created, the coverage can be used either to retrieve time series data from the raster bricks using `sits_get_data()` or to do the raster classification by calling the function `sits_classify`.

6.3 Classification using machine learning

There has been much recent interest in using classifiers such as support vector machines (Mountrakis, Im, and Ogole 2011) and random forests (Belgiu and Dragut 2016) for remote sensing images. Most often, researchers use a *space-first, time-later* approach, in which the dimension of the decision space is limited to the number of spectral bands or their transformations. Sometimes, the decision space is extended with temporal attributes. To do this, researchers filter the raw data to get smoother time series (Brown et al. 2013; Kastens et al. 2017). Then, using software such as TIMESAT (Jönsson and Eklundh 2004), they derive a small set of phenological parameters from vegetation indexes, like the beginning, peak, and length of the growing season (Estel et al. 2015; Pelletier et al. 2016).

In a recent review of machine learning methods to classify remote sensing data (Maxwell, Warner, and Fang 2018), the authors note that many factors influence the performance of these classifiers, including the size and quality of the training dataset, the dimension of the feature space, and the choice of the parameters. We support both *space-first, time-later* and *time-first, space-later* approaches. Therefore, the `sits` package provides functionality to explore the full depth of satellite image time series data.

When used in *time-first, space-later* approache, `sits` treats time series as a feature vector. To be consistent, the procedure aligns all time series from different years by its time proximity considering an given cropping schedule. Once aligned, the feature vector is formed by all pixel "bands". The idea is to have as many temporal attributes as possible, increasing the dimension of the classification space. In this scenario, statistical learning models are the natural candidates to deal with high-dimensional data: learning to distinguish all land cover and land use classes from trusted samples exemplars (the training data) to infer classes of a larger data set.

The SITS package provides a common interface to all machine learning models, using the `sits_train` function. this function takes two 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 the `sits_classify` function. In the examples that follow, we show how to apply each method for the classification of a single time series. Then, we disscuss how to classify full data cubes.

The following methods are available in SITS for training machine learning models:

- Linear discriminant analysis (`sits_lda`)
- Quadratic discriminant analysis (`sits_qda`)
- Multinomial logit and its variants 'lasso' and 'ridge' (`sits_mlr`)
- Support vector machines (`sits_svm`)
- Random forests (`sits_rfor`)
- Extreme gradient boosting (`sits_xgboost`)
- Deep learning (DL) using multi-layer perceptrons (`sits_deeplearning`)
- DL with 1D convolutional neural networks (`sits_CNN`),
- DL combining 1D CNN and multi-layer perceptron networks (`sits_tempCNN`)
- DL using 1D version of ResNet (`sits_ResNet`).
- DL using a combination of long-short term memory (LSTM) and 1D CNN (`sits_LSTM_FCN`)

For more details on each method, please see the vignette "Machine Learning for Data Cubes using the SITS package".

6.4 Cube classification

The continuous observation of the Earth surface provided by orbital sensors is unprecedented in history. Just for the sake of illustration, a unique tile from MOD13Q1 product, a square of 4800 pixels provided every 16 days since February 2000 takes around 18GB of uncompressed data to store only one band or vegetation index. This data deluge puts the field into a big data era and imposes challenges to design and build technologies that allow the Earth observation community to analyse those data sets (Câmara et al. 2017).

To classify a data cube, use the function `sits_classify()` as described below. This function works with cubes built from raster bricks. The classification algorithms allows users to choose how many process 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 small raster brick image that accompany the package.

6.4.1 Steps for cube classification

Once a data cube which has associated files is defined, the steps for classification are:

1. Select a set of training samples.
2. Train a machine learning model
3. Classify the data cubes using the model, producing a data cube with class probabilities.
4. Label the cube with probabilities, including data smoothing if desired.

6.4.2 Adjustments for improved performance

To reduce processing time, it is necessary to adjust `sits_classify()` according to the capabilities of the server. The package tries to keep memory use to a minimum, performing garbage collection to free memory as often as possible. Nevertheless, there is an inevitable 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 first parameter is `memsize`. It controls the size of the main memory (in GBytes) to be used for classification. The user must specify how much free memory will be available. The second factor controlling performance of raster classification is `multicores`. Once a block of data is read from disk into main memory, it is split into different cores, as specified by the user. In general, the more cores are assigned to classification, the faster the result will be. However, there are overheads in switching time, especially when the server has other processes running.

Based on current experience, the classification of a MODIS tile (4800 x 4800) with four bands and 400 time instances, covering 15 years of data, using SVM with a training data set of about 10,000 samples, takes about 24 hours using 20 cores and a memory size of 60 GB, in a server with 2.4GHz Xeon CPU and 96 GB of memory to produce the yearly classification maps.

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

#select a rfor model
xgb_model <- sits_train(samples_2bands, ml_method = sits_xgboost())
```

```
#> [1]  train-mlogloss:2.009944+0.000891    test-mlogloss:2.022905+0.001490
#> Multiple eval metrics are present. Will use test_mlogloss for early stopping.
#> Will train until test_mlogloss hasn't improved in 20 rounds.
#>
#> [11] train-mlogloss:0.620737+0.002764    test-mlogloss:0.755200+0.005152
#> [21] train-mlogloss:0.207436+0.001456    test-mlogloss:0.364395+0.009251
#> [31] train-mlogloss:0.094411+0.000752    test-mlogloss:0.251361+0.008440
#> [41] train-mlogloss:0.059334+0.000473    test-mlogloss:0.211024+0.009399
#> [51] train-mlogloss:0.048914+0.000651    test-mlogloss:0.197786+0.009918
#> [61] train-mlogloss:0.045721+0.000552    test-mlogloss:0.194586+0.010300
#> [71] train-mlogloss:0.044009+0.000838    test-mlogloss:0.192748+0.010751
```

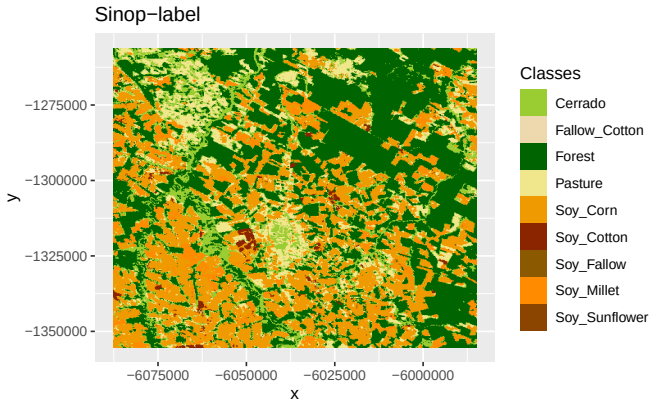
```
#> [81] train-mlogloss:0.043003+0.000582    test-mlogloss:0.191353+0.011066
#> [91] train-mlogloss:0.042301+0.000387    test-mlogloss:0.190409+0.011007
#> [100]    train-mlogloss:0.042005+0.000349    test-mlogloss:0.189919+0.011470
```

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

# create a raster metadata file based on the information about the files
sinop <- sits_cube(source = "LOCAL",
                   satellite = "TERRA",
                   sensor  = "MODIS",
                   name = "Sinop",
                   data_dir = data_dir,
                   parse_info = c("X1", "X2", "band", "date"))

# classify the raster image
sinop_probs <- sits_classify(sinop, ml_model = xgb_model,
                             memsize = 4, multicores = 1,
                             output_dir = tempdir())

# label the probability file
# (by default selecting the class with higher probability)
sinop_label <- sits_label_classification(sinop_probs, output_dir = tempdir())
plot(sinop_label, title = "Sinop-label")
```



6.5 Final remarks

Current approaches to image time series analysis still use limited number of attributes. A common approach is deriving a small set of phenological parameters from vegetation indices, like beginning, peak, and length of growing season (Brown et al. 2013), (Kastens et al. 2017), (Estel et al. 2015), (Pelletier et al. 2016). These phenological parameters are then fed in specialized classifiers such as TIMESAT (Jönsson and Eklundh 2004). These approaches do not use the

power of advanced statistical learning techniques to work on high-dimensional spaces with big training data sets (James et al. 2013).

Package `sits` can use the full depth of satellite image time series to create larger dimensional spaces. We tested different methods of extracting attributes from time series data, including those reported by Pelletier et al. (2016) and Kastens et al. (2017). Our conclusion is that part of the information in raw time series is lost after filtering. Thus, the method we developed uses all the data available in the time series samples. The idea is to have as many temporal attributes as possible, increasing the dimension of the classification space. Our experiments found out that modern statistical models such as support vector machines, and random forests perform better in high-dimensional spaces than in lower dimensional ones.

Chapter 7

Post classification smoothing using Bayesian techniques in SITS

This chapter describes a Bayesian smoothing method to reclassify the pixels, based on the machine learning probabilities. We consider that the output of the machine learning algorithm provides, for each pixel, the information on the probability of such pixel belonging to each of the target classes. Usually, we label a pixel as being of a given class if the associated class probability is higher than the probability of it belonging to any of the other classes. 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.

7.1 Introduction

Image classification post-processing has been defined as "a refinement of the labelling in a classified image in order to enhance its classification accuracy" (Huang et al. 2014). 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" (Schindler 2012): 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.

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. A traditional choice for smoothing classified images is the majority filter, where the class of the central pixel is replaced by the most frequent class of the neighbourhood. This technique is rather simplistic; more sophisticated methods use class probabilities. 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.

In this vignette, we introduce a Bayesian smoothing method, which provides the means to incorporate prior knowledge in data analysis. 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. As stated by [**?**]: "In the Bayesian paradigm, degrees of belief in states of nature are specified. Bayesian statistical methods start with existing 'prior' beliefs, and update these using data to give 'posterior' beliefs, which may be used as the basis for inferential decisions". Bayesian inference has now been established as a major method for assessing probability.

7.2 Overview of Bayesian estimattion

Most applications of machine learning methods for image classification use only the categorical result of the classifier which is the most probable class. The proposed method uses all class probabilities to compute our confidence in the result. 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.

7.2.1 Smmothing using Bayes' rule

Given the general principles of Bayesian inference, smoothing of classified images requires estimating the *likelihood* and the *prior* probability of each pixel belonging to each class. In order to express our problem in a more tractable form, we perform data transformations. More formally, consider a set of K classes that are candidates for labelling each pixel. Let $p_{i,k}$ be the probability of pixel i belonging to class k, $k = 1, \ldots, K$. We have

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

We label a pixel p_i as being of class k if

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

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

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

where $p_{i,k}$ is the probability of class k. 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 neighbours and reduced the variance of the estimated class for each pixel.

We further make the transformation

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

which measures the *logit* (log of the odds) associated to classifying the pixel i as being of class k. The support of $x_{i,k}$ is $\mathbb{R}$. Let V_i be a spatial neighbourhood for pixel i. We use Bayes' rule to update the value $x_{i,k}$ based on the neighbourhood, assuming independence between the classes. In this way, the update is performed for each class k at a time.

For each pixel, the random variable that describes the class probability is denoted by $\theta_{i,k}$. Therefore, we can express Bayes' rule for each combination of pixel and class as

$$\pi(\theta_{i,k}|x_{i,k}) \propto \pi(x_{i,k}|\theta_{i,k})\pi(\theta_{i,k}).$$

We assume the prior distribution $\pi(\theta_{i,k})$ and the likelihood $\pi(x_{i,k}|\theta_{i,k})$ are modelled by Gaussian distributions. In this case, the posterior will also be a Gaussian distribution. To estimate the prior distribution for a pixel, we consider that all pixels in the spatial neighbourhood V_i of pixel i follow the same Gaussian distribution with parameters $m_{i,k}$ and $s^2_{i,k}$. Thus, the prior is expressed as

$$\theta_{i,k} \sim N(m_{i,k}, s^2_{i,k}).$$

In the above equation, the parameter $m_{i,k}$ is the local mean of the probability distribution of values for class k and $s^2_{i,k}$ is the local variance for class k. We estimate the local mean and variance by considering the neighbouring pixels in space. Let $\#(V_i)$ be the number of elements in the spatial neighbourhood V_i. The local mean is calculated by:

$$m_{i,k} = \frac{\sum_{j \in V_i} x_{j,k}}{\#(V_i)}$$

and the local variance by

$$s^2_{i,k} = \frac{\sum_{j \in V_i} [x_{j,k} - m_{i,k}]^2}{\#(V_i) - 1}.$$

We also consider that the likelihood follows a normal distribution. We take the likelihood as being the distribution of $x_{i,k}$, conditioned by the local variable $\theta_{i,k}$. This conditional distribution is also taken as normal with parameters $\theta_{i,k}$ and σ^2_k, expressed as

$$x_{i,k}|\theta_{i,k} \sim N(\theta_{i,k}, \sigma^2_k)$$

In the likelihood equation above, σ^2_k is a hyper-parameter that controls the level of smoothness.The Bayesian smoothing estimates the value of $\theta_{i,k}$ conditioned by the data $x_{i,k}$. This is the updated value of the logit of class probability for class k of pixel i. Since both the prior and the likelihood are assumed as Gaussian distribution, based on Bayesian statistics the value of conditional mean for a normal distribution is given by:

$$E[\theta_{i,k}|x_{i,k}] = \frac{m_{i,t} \times \sigma^2_k + x_{i,k} \times s^2_{i,k}}{\sigma^2_k + s^2_{i,k}}$$

which can also be expressed as

$$E[\theta_{i,k}|x_{i,k}] = \left[\frac{s_{i,k}^2}{\sigma_k^2 + s_{i,k}^2}\right] \times x_{i,k} + \left[\frac{\sigma_k^2}{\sigma_k^2 + s_{i,k}^2}\right] \times m_{i,k}$$

The updated value for the class probability of the pixel is a weighted average between the original logit value $x_{i,k}$ and the mean of the class logits $m_{i,k}$ for the neighboring pixels. When the local class variance of the neighbors $s_{i,k}^2$ is high relative to the smoothing factor σ_k^2, 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 variance $s_{i,k}^2$ decreases relative to the smoothness factor σ_k^2, 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.

The hyperparameter σ_k^2 sets the level of smoothness. If σ_k^2 is zero, the smoothed value $E[\mu_{i,,k}|l_{i,k}]$ is equal to the pixel value $l_{i,k}$. Higher values of σ_k^2 will cause the assignment of the local mean to the pixel updated probability. In practice, σ_k^2 is a user-controlled parameter that will be set by users based on their knowledge of the region to be classified. In our case, after some classification tests, we decided to set the parameters V as the Moore neighborhood where each pixel is connected to all those pixels with Chebyshev distance of 1, and $\sigma_k^2 = 20$ for all k. This level of smoothness showed the best performance in the technical validation.

7.3 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 more probability images, one per requested clasification interval. The next step is to apply the `sits_label_classification` 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 smooting, it suffices to include the `smoothing = bayesian` parameter. If desired, the `variance` parameter (associated to the hyperparameter σ_k^2 described above) can control the degree of smoothness.

```
# Retrieve the data for the Mato Grosso state
data("samples_matogrosso_mod13q1")

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

```
#select a rfor model
xgb_model <- sits_train(samples_2bands, ml_method = sits_xgboost())
```

```
#> [1]  train-mlogloss:2.009944+0.000891    test-mlogloss:2.022905+0.001490
#> Multiple eval metrics are present. Will use test_mlogloss for early stopping.
#> Will train until test_mlogloss hasn't improved in 20 rounds.
#>
#> [11] train-mlogloss:0.620737+0.002764    test-mlogloss:0.755200+0.005152
#> [21] train-mlogloss:0.207436+0.001456    test-mlogloss:0.364395+0.009251
#> [31] train-mlogloss:0.094411+0.000752    test-mlogloss:0.251361+0.008440
#> [41] train-mlogloss:0.059334+0.000473    test-mlogloss:0.211024+0.009399
#> [51] train-mlogloss:0.048914+0.000651    test-mlogloss:0.197786+0.009918
#> [61] train-mlogloss:0.045721+0.000552    test-mlogloss:0.194586+0.010300
#> [71] train-mlogloss:0.044009+0.000838    test-mlogloss:0.192748+0.010751
#> [81] train-mlogloss:0.043003+0.000582    test-mlogloss:0.191353+0.011066
#> [91] train-mlogloss:0.042301+0.000387    test-mlogloss:0.190409+0.011007
#> [100]    train-mlogloss:0.042005+0.000349    test-mlogloss:0.189919+0.011470
```

```
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",
                  satellite = "TERRA",
                  sensor  = "MODIS",
                  name = "Sinop",
                  data_dir = data_dir,
                  parse_info = c("X1", "X2", "band", "date"),
)

# classify the raster image and generate a probability file
raster_probs <- sits_classify(raster_cube, ml_model = xgb_model,
                              memsize = 4, multicores = 2,
                              output_dir = tempdir())

# smooth the result with a bayesian filter
raster_probs_bayes <- sits_smooth(raster_probs, output_dir = tempdir())

# label the smoothed probability images
raster_class <- sits_label_classification(raster_probs_bayes, output_dir = tempdir())
```

The result is shown below.

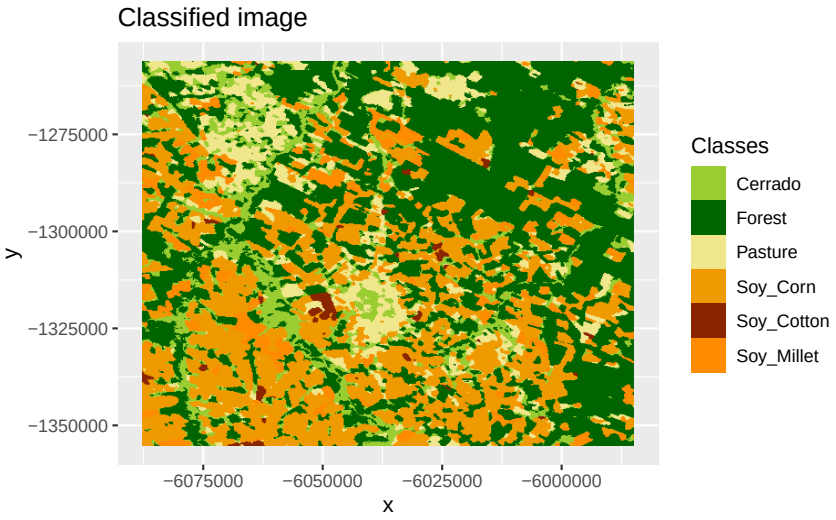


Figure 7.1: Classified image post-processed with Bayesian smoothing. The image coordinates (*meters*) shown at vertical and horizontal axis are in MODIS sinusoidal projection.

Chapter 8

Validation and accuracy measurements in SITS

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

8.1 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 (Hastie, Tibshirani, and J. 2009).

To be sure, 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 Hastie, Tibshirani, and J. (2009), 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 (Hastie, Tibshirani, and J. 2009), 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
prediction.mx <- sits_kfold_validate(cerrado_2classes,
                                     folds = 5,
                                     ml_method = sits_rfor())
```

8.2 Comparing different validation methods

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.

```
# 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()

# adjust the multicores parameters to suit your machine

## SVM model
conf_svm.tb <- sits_kfold_validate(
    samples_matogrosso_mod13q1,
    folds = 5,
    multicores = 2,
    ml_method = sits_svm(kernel = "radial", cost = 10))

# Give a name to the SVM model
conf_svm.tb$name <- "svm_10"

# store the result
results[[length(results) + 1]] <- conf_svm.tb
```

```
conf_rfor.tb <- sits_kfold_validate(
    samples_matogrosso_mod13q1,
    folds = 5,
    multicores = 1,
    ml_method = sits_rfor(num_trees = 500))

# Give a name to the model
conf_rfor.tb$name <- "rfor_500"

# store the results in a list
results[[length(results) + 1]] <- conf_rfor.tb

# Save to an XLS file
sits_to_xlsx(results, file = tempfile(fileext = ".xlsx"))
```

```
#> Saved Excel file /tmp/Rtmp1jIy7f/file985915b58a58.xlsx
```

Accardo, Agostino, M Affinito, M Carrozzi, and F Bouquet. 1997. "Use of the Fractal Dimension for the Analysis of Electroencephalographic Time Series." *Biological Cybernetics* 77 (5): 339–50.

Aghabozorgi, Saeed, Ali Seyed Shirkhorshidi, and Teh Ying Wah. 2015. "Time-Series Clustering–a Decade Review." *Information Systems* 53: 16–38.

Appel, Marius, and Edzer Pebesma. 2019. "On-Demand Processing of Data Cubes from Satellite Image Collections with the Gdalcubes Library." *Data* 4 (92).

Arvor, D., M. Meirelles, V. Dubreuil, A. Bégué, and Y. E. Shimabukuro. 2012. "Analyzing the Agricultural Transition in Mato Grosso, Brazil, Using Satellite-Derived Indices." *Applied Geography* 32 (2): 702–13.

Atkinson, Peter M, C Jeganathan, Jadu Dash, and Clement Atzberger. 2012. "Inter-Comparison of Four Models for Smoothing Satellite Sensor Time-Series Data to Estimate Vegetation Phenology." *Remote Sensing of Environment* 123: 400–417.

Atzberger, Clement, and Paul HC Eilers. 2011. "Evaluating the Effectiveness of Smoothing Algorithms in the Absence of Ground Reference Measurements." *International Journal of Remote Sensing* 32 (13): 3689–3709.

Belgiu, Mariana, and Lucian Dragut. 2016. "Random Forest in Remote Sensing: A Review of Applications and Future Directions." *ISPRS Journal of Photogrammetry and Remote Sensing* 114: 24–31.

Bradley, Bethany A, Robert W Jacob, John F Hermance, and John F Mustard. 2007. "A Curve Fitting Procedure to Derive Inter-Annual Phenologies from Time Series of Noisy Satellite NDVI Data." *Remote Sensing of Environment* 106 (2): 137–45.

Brown, J. Christopher, Jude H. Kastens, Alexandre Camargo Coutinho, Daniel de Castro Victoria, and Christopher R. Bishop. 2013. "Classifying Multiyear Agricultural Land Use Data from Mato Grosso Using Time-Series MODIS Vegetation Index Data." *Remote Sensing of Environment* 130: 39–50.

Câmara, Gilberto, Gilberto Queiroz, Lubia Vinhas, Karine Ferreira, Ricardo Cartaxo, Rolf Simoes, Eduardo Llapa, Luiz Assis, and Alber Sanchez. 2017. "The E-Sensing Architecture for Big Earth Observation Data Analysis." In *Big Data from Space (Bids'17)*, 48–51.

Chang, Chih-Chung, and Chih-Jen Lin. 2011. "LIBSVM: A Library for Support Vector Machines." *ACM Transactions on Intelligent Systems and Technology (TIST)* 2 (3): 27.

Chen, Jin, Per. Jönsson, Masayuki Tamura, Zhihui Gu andBunkei Matsushita, and Lars Eklundh. 2004. "A Simple Method for Reconstructing a High-Quality NDVI Time-Series Data Set Based on the Savitzky–Golay Filter." *Remote Sensing of Environment* 91 (3-4): 332–44.

Chen, Tianqi, and Carlos Guestrin. 2016. "Xgboost: A Scalable Tree Boosting System." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–94.

Chollet, Francois, and J. J. Allaire. 2018. *Deep Learning with R*. New York, NY: Manning Publications.

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning* 20 (3): 273–97.

Efron, Bradley, and Trevor Hastie. 2016. *Computer Age Statistical Inference*. Vol. 5. Cambridge University Press.

Estel, Stephan, Tobias Kuemmerle, Camilo Alcantara, Christian Levers, Alexander Prishchepov, and Patrick Hostert. 2015. "Mapping Farmland Abandonment and Recultivation Across Europe Using MODIS NDVI Time Series." *Remote Sensing of Environment* 163: 312–25.

Fawaz, Hassan Ismail, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. "Deep Learning for Time Series Classification: A Review." *Data Mining and Knowledge Discovery* 33 (4): 917–63.

Galford, Gillian L, John F Mustard, Jerry Melillo, Aline Gendrin, Carlos C Cerri, and Carlos E Cerri. 2008. "Wavelet Analysis of MODIS Time Series to Detect Expansion and Intensification of Row-Crop Agriculture in Brazil." *Remote Sensing of Environment* 112 (2): 576–87.

Gomez, Cristina, Joanne C. White, and Michael A. Wulder. 2016. "Optical Remotely Sensed Time Series Data for Land Cover Classification: A Review." *{ISPRS} Journal of Photogrammetry and Remote Sensing* 116: 55–72.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Haralick, Robert M, Stanley R Sternberg, and Xinhua Zhuang. 1987. "Image Analysis Using Mathematical Morphology." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 4: 532–50.

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

Hennig, Christian. 2015. "Clustering Strategy and Method Selection." In *Handbook of Cluster Analysis*, edited by Christian Hennig, Marina Meila, Fionn Murtagh, and Roberto Rocci. CRC Press.

Hird, Jennifer N, and Gregory J McDermid. 2009. "Noise Reduction of NDVI Time Series: An Empirical Comparison of Selected Techniques." *Remote Sensing of Environment* 113 (1): 248–58.

Hochreiter, Sepp. 1998. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions." *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (02): 107–16.

Huang, Xin, Qikai Lu, Liangpei Zhang, and Antonio Plaza. 2014. "New Postprocessing Methods for Remote Sensing Image Classification: A Systematic Study." *IEEE Transactions on Geoscience and Remote Sensing* 52 (11): 7140–59.

Hubert, Lawrence, and Phipps Arabie. 1985. "Comparing Partitions." *Journal of Classification* 2 (1): 193–218.

INPE. 2017. "Amazon Deforestation Monitoring Project (PRODES)." (National Institute for Space Research, Brazil). www.obt.inpe.br/prodes.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *arXiv Preprint arXiv:1502.03167*.

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

Jönsson, Per, and Lars Eklundh. 2004. "TIMESAT—a Program for Analyzing Time-Series of Satellite Sensor Data." *Computers & Geosciences* 30 (8): 833–45.

Karim, Fazle, Somshubra Majumdar, Houshang Darabi, and Shun Chen. 2018. "LSTM Fully Convolutional Networks for Time Series Classification." *IEEE Access* 6: 1662–9.

Karim, Fazle, Somshubra Majumdar, Houshang Darabi, and Samuel Harford. 2019. "Multivariate LSTM-FCNS for Time Series Classification." *Neural Networks* 116: 237–45.

Kastens, J., J. Brown, A. Coutinho, C. Bishop, and J. Esquerdo. 2017. "Soy Moratorium Impacts on Soybean and Deforestation Dynamics in Mato Grosso, Brazil." *PLOS ONE* 12 (4): e0176168.

Kennedy, Robert E., Zhiqiang Yang, and Warren B. Cohen. 2010. "Detecting Trends in Forest Disturbance and Recovery Using Yearly Landsat Time Series."

Remote Sensing of Environment 114 (12): 2897–2910.

Keogh, Eamonn, Jessica Lin, and Wagner Truppel. 2003. "Clustering of Time Series Subsequences Is Meaningless: Implications for Previous and Future Research." In *Data Mining, 2003. ICDM 2003. Third Ieee International Conference on*, 115–22.

Lambin, E. F., and M. Linderman. 2006. "Time Series of Remote Sensing Data for Land Change Science." *IEEE Transactions on Geoscience and Remote Sensing* 44 (7): 1926–8.

Lambin, Eric F, Helmut J Geist, and Erika Lepers. 2003. "Dynamics of Land-Use and Land-Cover Change in Tropical Regions." *Annual Review of Environment and Resources* 28 (1): 205–41.

Madden, Hannibal H. 1978. "Comments on the Savitzky-Golay Convolution Method for Least-Squares-Fit Smoothing and Differentiation of Digital Data." *Analytical Chemistry* 50 (9): 1383–6.

Maus, Victor, Gilberto Camara, Ricardo Cartaxo, Alber Sanchez, Fernando M Ramos, and Gilberto R de Queiroz. 2016. "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* 9 (8): 3729–39.

Maus, Victor, Gilberto Câmara, Marius Appel, and Edzer Pebesma. 2019. "DtwSat: Time-Weighted Dynamic Time Warping for Satellite Image Time Series Analysis in R." *Journal of Statistical Software* 88 (5): 1–31.

Maxwell, Aaron E., Timothy A. Warner, and Fang Fang. 2018. "Implementation of Machine-Learning Classification in Remote Sensing: An Applied Review." *International Journal of Remote Sensing* 39 (9): 2784–2817. https://doi.org/10.1080/01431161.2018.1433343.

Mountrakis, G., J. Im, and C. Ogole. 2011. "Support Vector Machines in Remote Sensing: A Review." *ISPRS Journal of Photogrammetry and Remote Sensing* 66 (3): 247–59.

Pasquarella, Valerie J., Christopher E. Holden, Les Kaufman, and Curtis E. Woodcock. 2016. "From Imagery to Ecology: Leveraging Time Series of All Available LANDSAT Observations to Map and Monitor Ecosystem State and Dynamics." *Remote Sensing in Ecology and Conservation* 2 (3): 152–70. https://doi.org/10.1002/rse2.24.

Pelletier, Charlotte, Silvia Valero, Jordi Inglada, Nicolas Champion, and Gerard Dedieu. 2016. "Assessing the Robustness of Random Forests to Map Land Cover with High Resolution Satellite Image Time Series over Large Areas." *Remote Sensing of Environment* 187: 156–68.

Pelletier, Charlotte, Geoffrey I Webb, and François Petitjean. 2019. "Temporal Convolutional Neural Network for the Classification of Satellite Image Time

Series." *Remote Sensing* 11 (5): 523.

Petitjean, F., J. Inglada, and P. Gancarski. 2012. "Satellite Image Time Series Analysis Under Time Warping." *IEEE Transactions on Geoscience and Remote Sensing* 50 (8): 3081–95. https://doi.org/10.1109/TGRS.2011.2179050.

Petitjean, François, Jordi Inglada, and Pierre Gançarskv. 2011. "Clustering of Satellite Image Time Series Under Time Warping." In *Analysis of Multi-Temporal Remote Sensing Images (Multi-Temp), 2011 6th International Workshop on the*, 69–72. IEEE.

Picoli, Michelle Cristina Araujo, Gilberto Camara, Ieda Sanches, Rolf Simões, Alexandre Carvalho, Adeline Maciel, Alexandre Coutinho, et al. 2018. "Big Earth Observation Time Series Analysis for Monitoring Brazilian Agriculture." *ISPRS Journal of Photogrammetry and Remote Sensing* 145: 328–39.

Ruder, Sebastian. 2016. "An Overview of Gradient Descent Optimization Algorithms." *arXiv Preprint arXiv:1609.04747*.

Rußwurm, Marc, and Marco Korner. 2017. "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*, 11–19.

Rußwurm, Marc, and Marco Körner. 2018. "Multi-Temporal Land Cover Classification with Sequential Recurrent Encoders." *ISPRS International Journal of Geo-Information* 7 (4): 129.

Sakamoto, Toshihiro, Masayuki Yokozawa, Hitoshi Toritani, Michio Shibayama, Naoki Ishitsuka, and Hiroyuki Ohno. 2005. "A Crop Phenology Detection Method Using Time-Series MODIS Data." *Remote Sensing of Environment* 96 (3-4): 366–74.

Schindler, Konrad. 2012. "An Overview and Comparison of Smooth Labeling Methods for Land-Cover Classification." *IEEE Transactions on Geoscience and Remote Sensing* 50 (11): 4534–45.

Shao, Yang, Ross S. Lunetta, Brandon Wheeler, John S. Iiames, and James B. Campbell. 2016. "An Evaluation of Time-Series Smoothing Algorithms for Land-Cover Classifications Using MODIS-NDVI Multi-Temporal Data." *Remote Sensing of Environment* 174: 258–65.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *The Journal of Machine Learning Research* 15 (1): 1929–58.

Vávra, F., P. Nový, H. Masková, M. Kotlíková, and A. Netrvalová. 2004. "Morphological Filtration for Time Series." In *Proceedings of the 3rd International Conference Aplimat*, 983–89. Bratislava, Slovakia: Slovak University of Technology in Bratislava.

Verbesselt, Jan, Rob Hyndman, Glenn Newnham, and Darius Culvenor. 2010. "Detecting Trend and Seasonal Changes in Satellite Image Time Series." *Remote Sensing of Environment* 114 (1): 106–15.

Wang, Zhiguang, Weizhong Yan, and Tim Oates. 2017. "Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline." In *2017 International Joint Conference on Neural Networks (Ijcnn)*, 1578–85. IEEE.

Ward, Joe H. 1963. "Hierarchical Grouping to Optimize an Objective Function." *Journal of the American Statistical Association* 58 (301): 236–44.

Whittaker, Edmund T. 1922. "On a New Method of Graduation." *Proceedings of the Edinburgh Mathematical Society* 41: 63–75.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.

Wright, Marvin, and Andreas Ziegler. 2017. "Ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R." *Journal of Statistical Software* 77 (1): 1–17. https://doi.org/10.18637/jss.v077.i01.

Zhou, Jie, Li Jia, Massimo Menenti, and Ben Gorte. 2016. "On the Performance of Remote Sensing Time Series Reconstruction Methods – a Spatial Comparison." *Remote Sensing of Environment* 187: 367–84.