# Machine Learning for Data Cubes using the SITS package

**Rolf Simoes** *National Institute for Space Research (INPE), Brazil*
**Gilberto Camara** *National Institute for Space Research (INPE), Brazil*
**Alexandre Carvalho** *Institute for Applied Economics Research (IPEA), Brazil*
**Pedro R. Andrade** *National Institute for Space Research (INPE), Brazil*
**Victor Maus** *University of Vienna*

This vignette 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 learining methods avilable in SITS include linear and quadratic discrimination analysis, support vector machines, random forests, deep learning and neural networks.

## 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 et al., 2019] . In a recent review of machine learning methods to classify remote sensing data [Maxwell et al., 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.

## 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 "inSitu", which is downloadable from the website associated to the "e-sensing" project. The examples below use four out of six bands ("ndvi", "evi", "nir", "mir")

## Visualizing Samples

One useful way of describing and understanding the samples is by plotting them. A direct way of doing so is using the `sits_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 "Forest", "Pasture" and "Cerrado" labels in the "evi" band.

In the above plots, 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 `sits_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.

## 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 example illustrate 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 `sits_plot`.

```
# Train a machine learning model for the mato grosso dataset using SVM
# The parameters are those of the "e1071:svm" method
model <- sits_train(data = mato_grosso_4bands, ml_method = sits_rfor())
# Classify using random forest model and plot the result
class.tb <- sits_classify(point_mt_4bands, 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
sits_plot(class.tb)
```

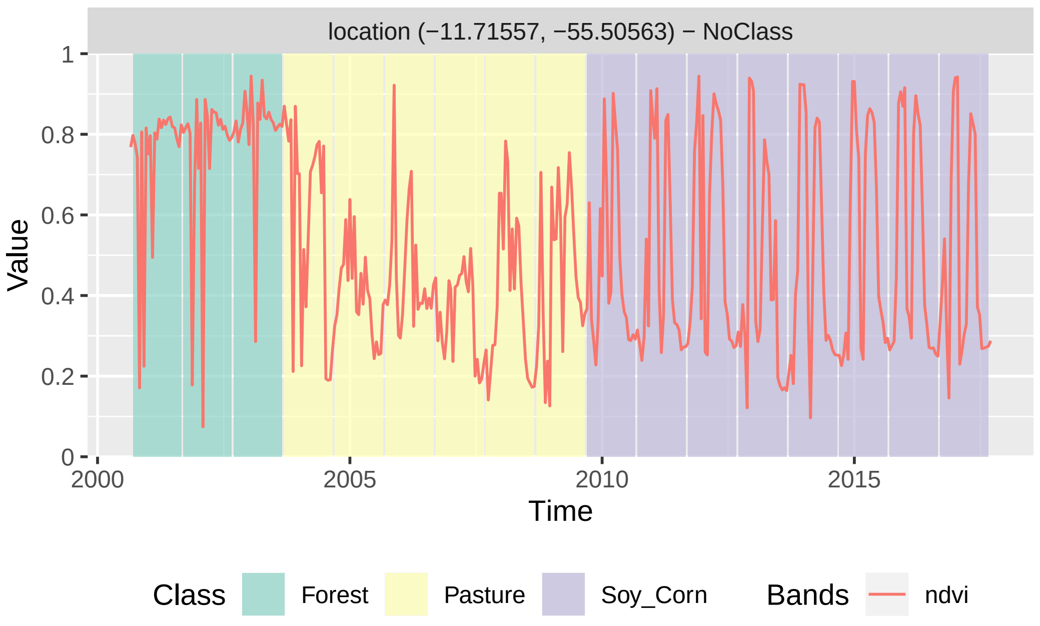

Figure 1: SVM classification of a 16 years time series. The location (latitude, longitude) shown at the top of the graph is in geographic coordinate system (WGS84 *datum*).

### 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 et al., 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(mato_grosso_4bands, ml_method = sits_svm(kernel = "radial", cost =
# Classify using SVM model and plot the result
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.25, bands_suffix = "") %>%
    sits_classify(svm_model) %>%
    sits_plot()
```

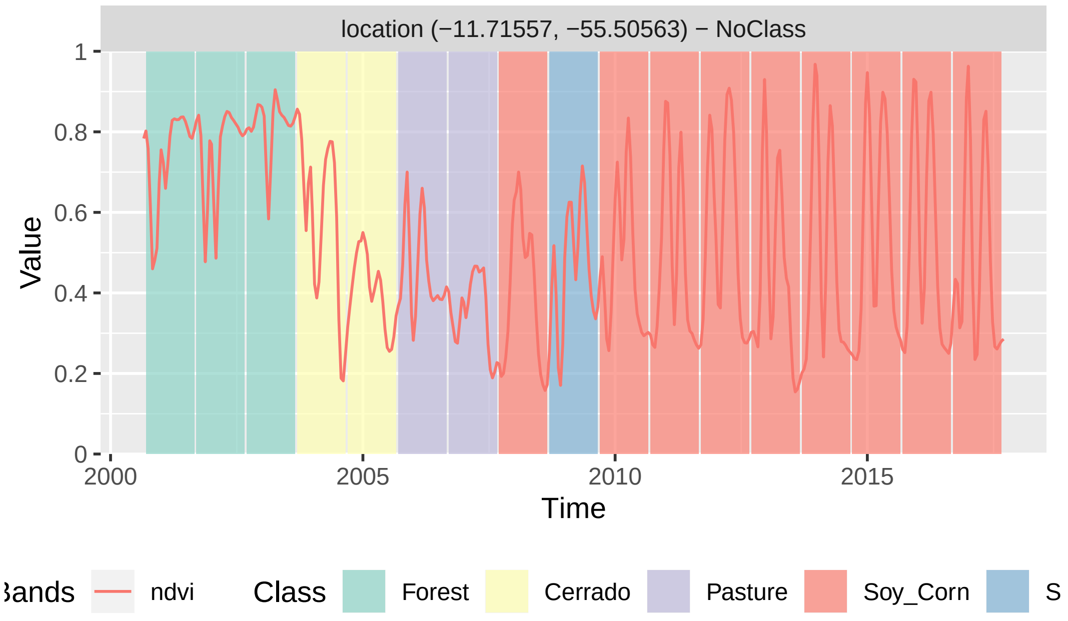

Figure 2: SVM classification of a 16 years time series. The location (latitude, longitude) shown at the top of the graph is in geographic coordinate system (WGS84 *datum*).

```
# 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 Cerrado
##  5 2004-09-13 2005-08-29 Cerrado
##  6 2005-09-14 2006-08-29 Pasture
##  7 2006-09-14 2007-08-29 Pasture
##  8 2007-09-14 2008-08-28 Soy_Corn
##  9 2008-09-13 2009-08-29 Soy_Millet
## 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 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 sensitivity to the input parameters points to a limitation when using the SVM method for classifying time series.

## 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`.

```r
# Retrieve the set of samples (provided by EMBRAPA) from the
# Mato Grosso region for train the Random Forest model.
rfor_model <- sits_train(mato_grosso_4bands, sits_rfor(num_trees = 500))
# Classify using Random Forest model and plot the result
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(rfor_model) %>%
    sits_plot()
```

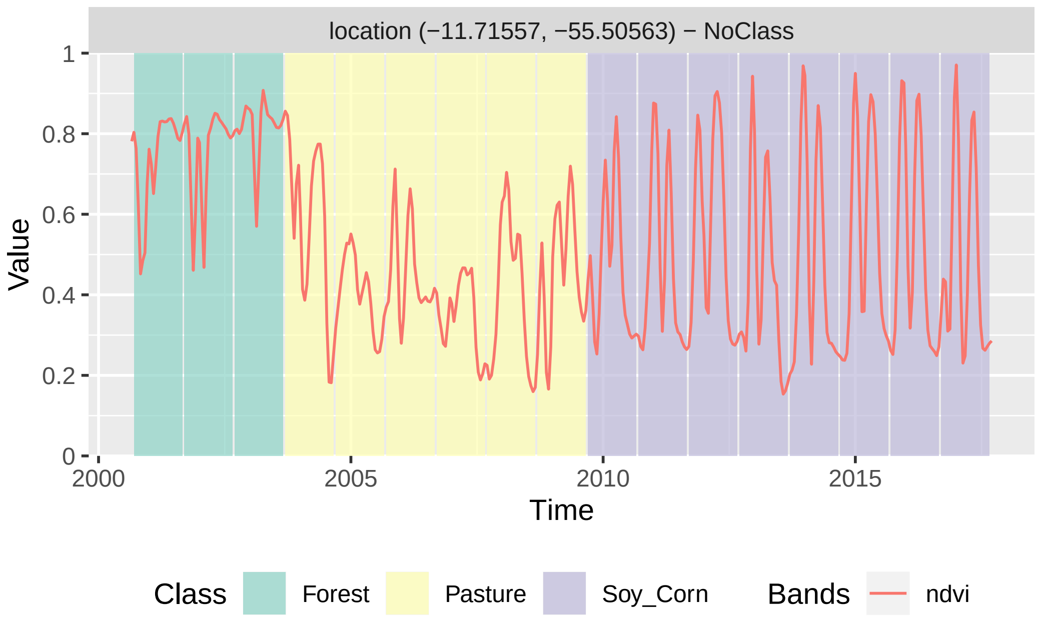

Figure 3: Random forest classification of a 16 years time series. The location (latitude, longitude) shown at the top of the graph are in geographic coordinate system (WGS84 *datum*).

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 et al., 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 et al., 2009]. For this reason, in classification comparisons, its performance tends to be weaker than methods such as extreme gradient boosting, described below. Nevertheless, random forest classifiers can be quite useful to provide a baseline to compare with more sophisticated methods.

### 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.

```r
# Train a machine learning model for the mato grosso dataset using SVM
# The parameters are those of the "xgboost" package
xgb_model <- sits_train(mato_grosso_4bands, sits_xgboost(eta = 0.3,
                                                        gamma = 0.01,
                                                        max_depth = 6,
                                                        subsample = 0.9,
                                                        nrounds = 100,
                                                    verbose = FALSE))
# Classify and plot the result
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(xgb_model) %>%
    sits_plot()
```

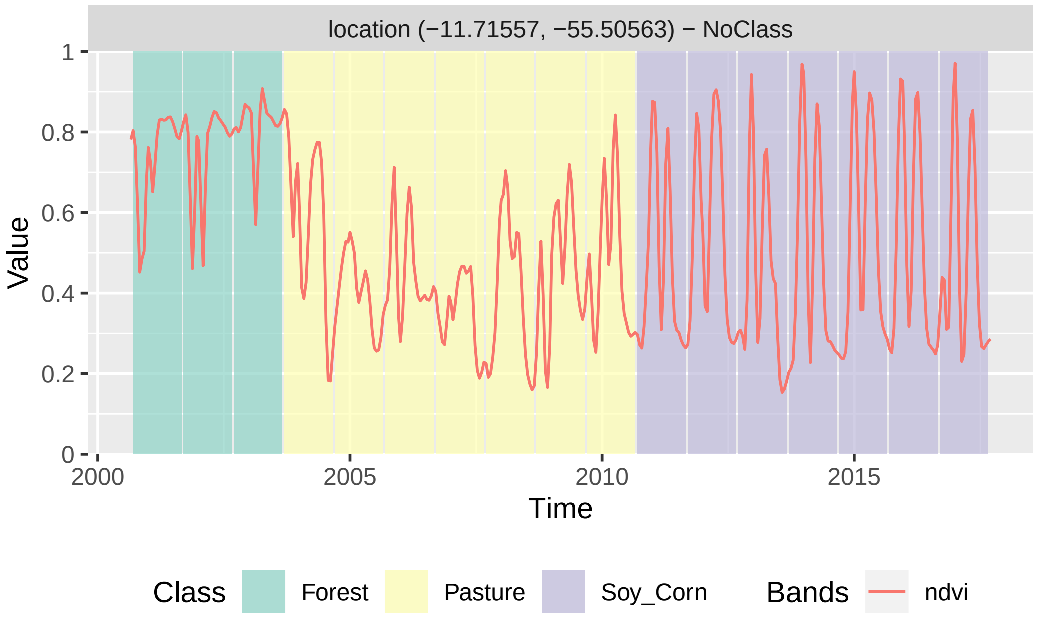

Figure 4: XGBoost classification of a 16 years time series. The location (latitude, longitude) shown at the top of the graph is in geographic coordinate system (WGS84 *datum*).

### 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 methods 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 $\theta$ 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 et al., 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 et al., 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 deep learning 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 et al., 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 a simple example of the `deep learning` method, for fast processing: (a) Two layers with 512 neurons each; (b) Using the 'elu' activation function and 'optimizer_adam'; (c) dropout rates of 40% and 30% for the layers.

```r
# train a machine learning model for the Mato Grosso data using an MLP
mlp_model <- sits_train(mato_grosso_4bands,
                       sits_deeplearning(
                         units            = c(512, 512, 512),
                         activation       = "elu",
                         dropout_rates    = c(0.50, 0.40, 0.30),
                         optimizer        = keras::optimizer_adam(lr = 0.001),
                         epochs           = 150,
                         batch_size       = 128,
                         validation_split = 0.2) )

# show training evolution
plot(environment(mlp_model)$history)
```

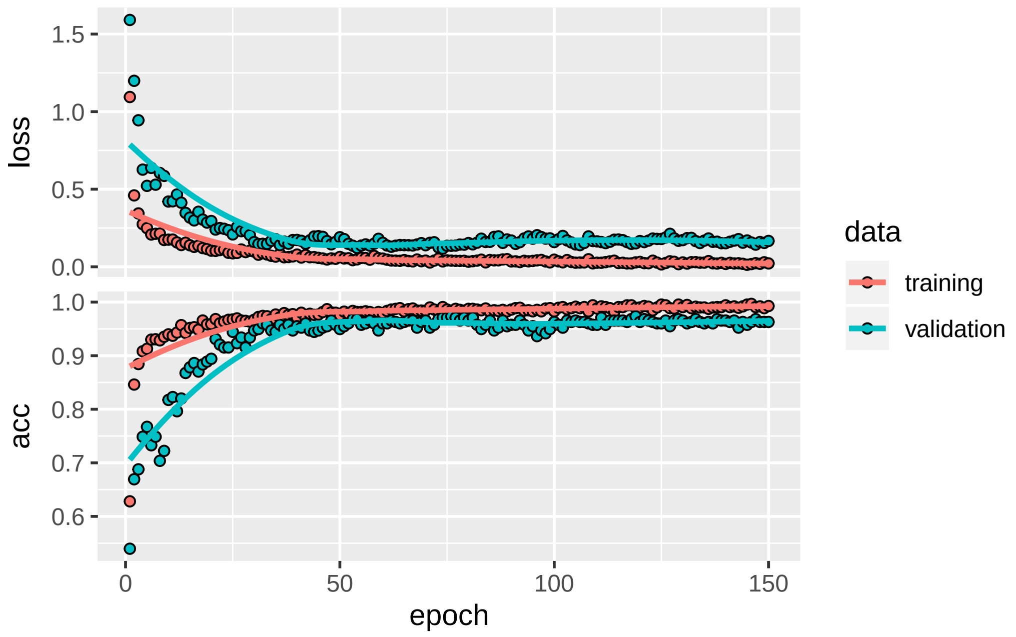

Figure 5: Multi-layer perceptron classification of a 16 year time series.

```r
# classify and plot
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(mlp_model) %>%
    sits_plot()
```

## 1D convolutional neural networks

Con

```r
# Retrieve the set of samples (provided by EMBRAPA) from the
# Mato Grosso region for train the  model.
data(samples_mt_ndvi)
# train a machine learning model using deep learning
cnn_model <- sits_train(mato_grosso_4bands,
                       sits_CNN(
                         units              = c(32, 32, 32),
                         kernels            = c(11, 7, 3),
                         activation         = 'relu',
                         L2_rate            = 1e-06,
                         dropout_rates      = c(0.50, 0.45, 0.40),
                         batch_normalization = FALSE,
                         optimizer          = keras::optimizer_adam(lr = 0.001),
                         epochs             = 150,
```

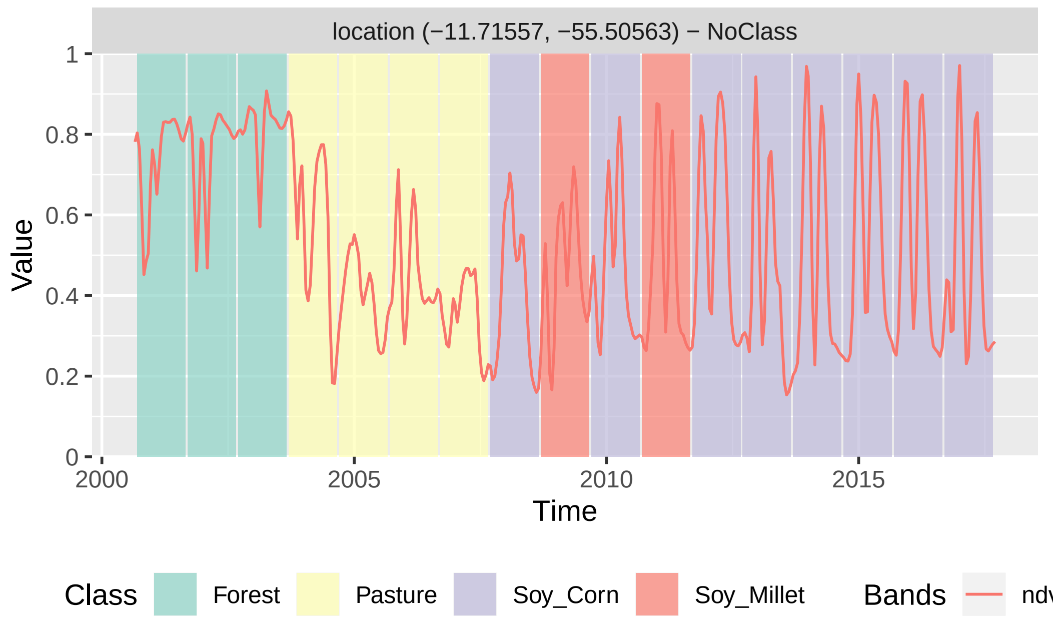

Figure 6: Multi-layer perceptron classification of a 16 year time series.

```
                         batch_size         = 128,
                         verbose            = 1,
                         binary_classification = FALSE) )

# show training evolution
plot(environment(cnn_model)$history)
```

```
# get a point to be classified
# classify and plot
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(cnn_model) %>%
    sits_plot()
```

### 1D CNN and multi-layer perceptron networks

```
# train a machine learning model using tempCNN
tCNN_model <- sits_train(mato_grosso_4bands,
                       sits_tempCNN(
                          conv_units          = c(64, 64, 64),
                          conv_kernels        = c(7, 7, 7),
                          conv_activation     = 'relu',
                          conv_L2_rate        = 1e-06,
```

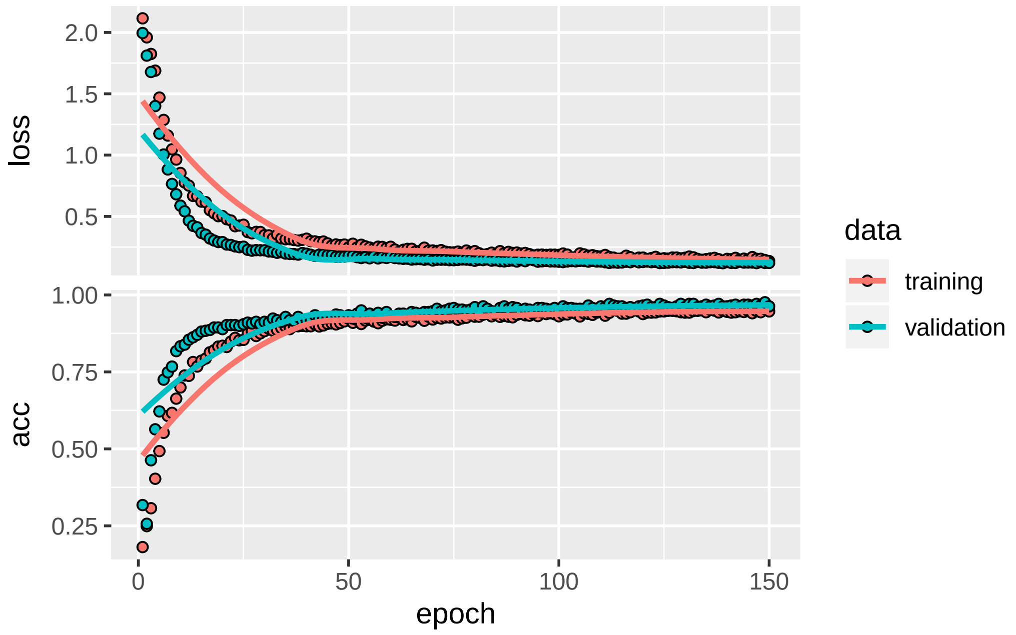

Figure 7: Deep learning classification of a 16 year time series. The location (latitude, longitude) shown at the top of the graph are in geographic coordinate system (WGS84 *datum*).

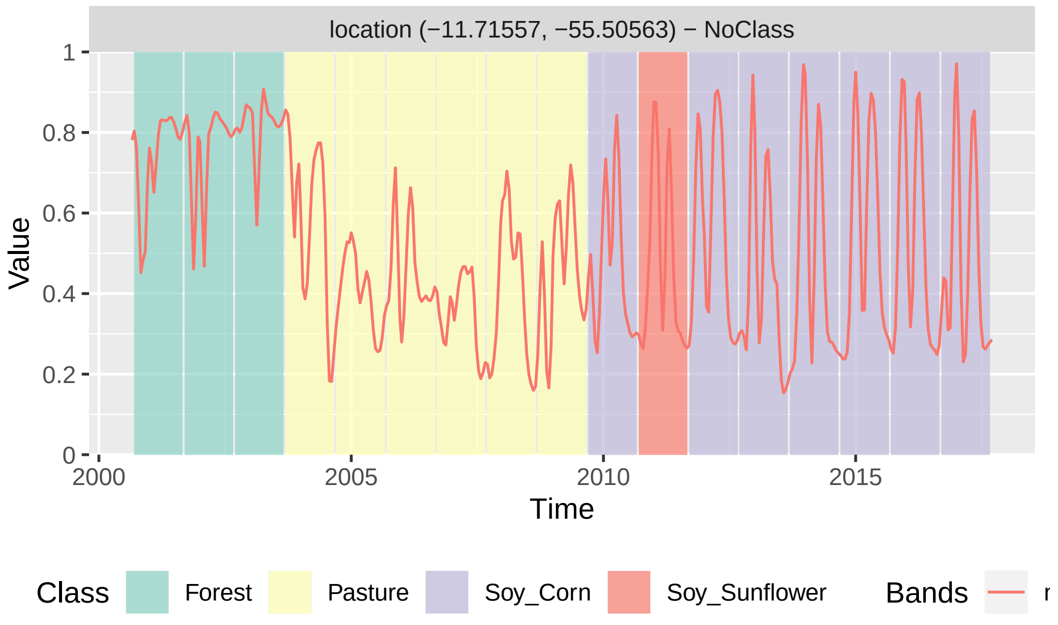

Figure 8: Deep learning classification of a 16 year time series. The location (latitude, longitude) shown at the top of the graph are in geographic coordinate system (WGS84 *datum*).

```
                                   conv_dropout_rates   = c(0.50, 0.50, 0.50),
                                   node_units           = c(512, 512, 512),
                                   node_activation      = 'elu',
                                   node_dropout_rates   = c(0.50, 0.45, 0.40),
                                   optimizer            = keras::optimizer_adam(lr = 0.001),
                                   epochs               = 150,
                                   batch_size           = 128,
                                   validation_split     = 0.2,
                                   verbose              = 1,
                                   binary_classification = FALSE) )

# show training evolution
plot(environment(tCNN_model)$history)
```

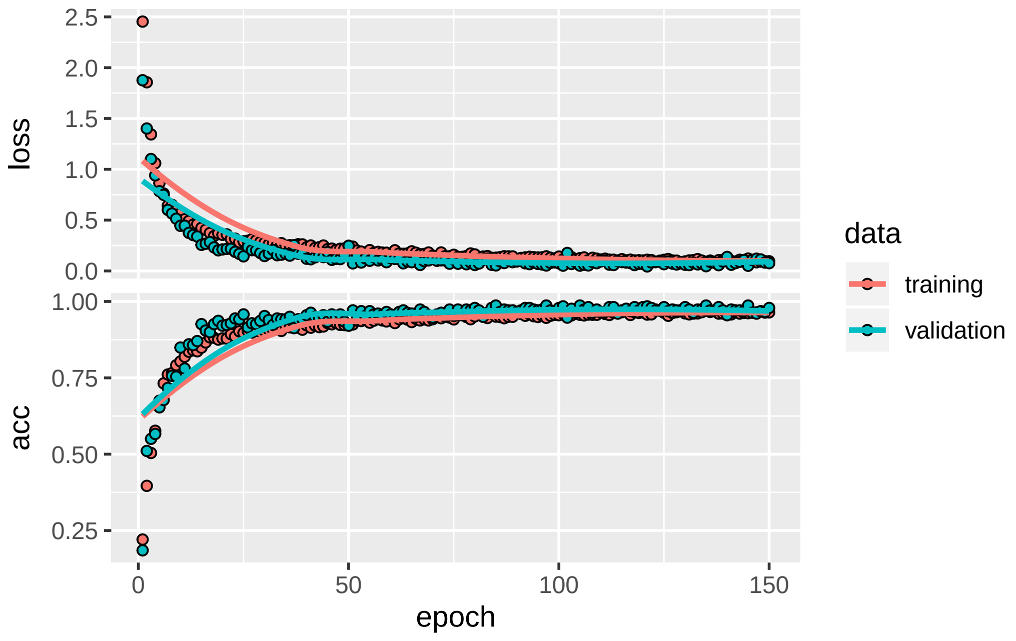

Figure 9: TempCNN classification of a 16 year time series.

```
# get a point to be classified
# classify and plot
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(tCNN_model) %>%
    sits_plot()
```

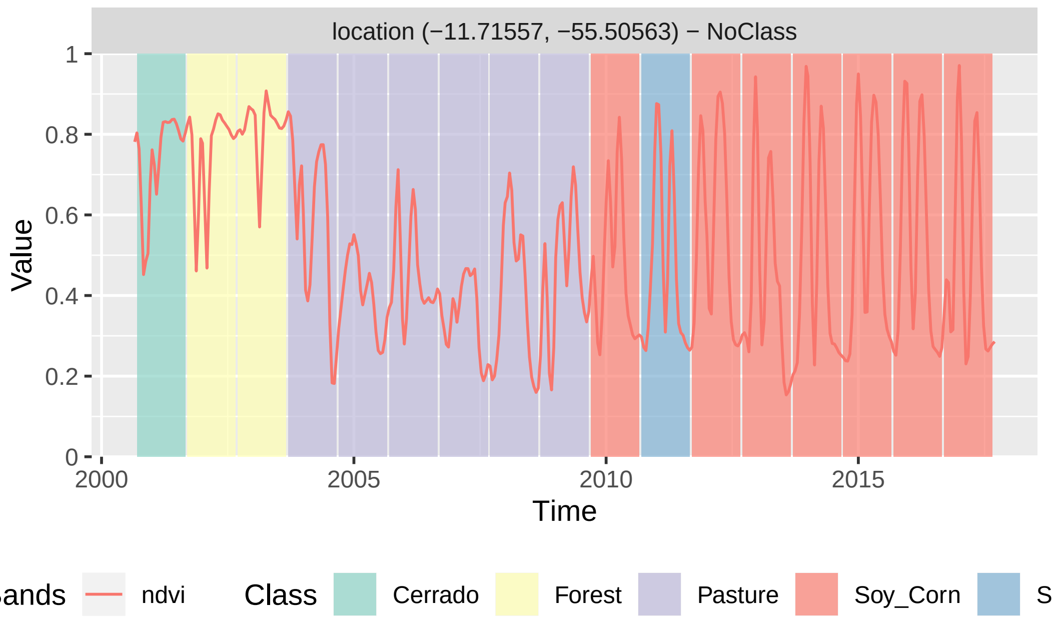

Figure 10: TempCNN classification of a 16 year time series.

### 1D version of ResNet

```
# train a machine learning model using tempCNN
resnet_model <- sits_train(mato_grosso_4bands,
                           sits_ResNet(
                               units                 = 16,
                               kernels               = c(11, 7, 3),
                               activation            = 'relu',
                               L2_rate               = 1e-06,
                               optimizer             = keras::optimizer_adam(lr = 0.001),
                               epochs                = 150,
                               batch_size            = 128,
                               validation_split      = 0.2,
                               verbose               = 1,
                               binary_classification = FALSE) )

# show training evolution
plot(environment(resnet_model)$history)
```

```
# get a point to be classified
# classify and plot
class.tb <- point_mt_4bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(resnet_model) %>%
    sits_plot()
```

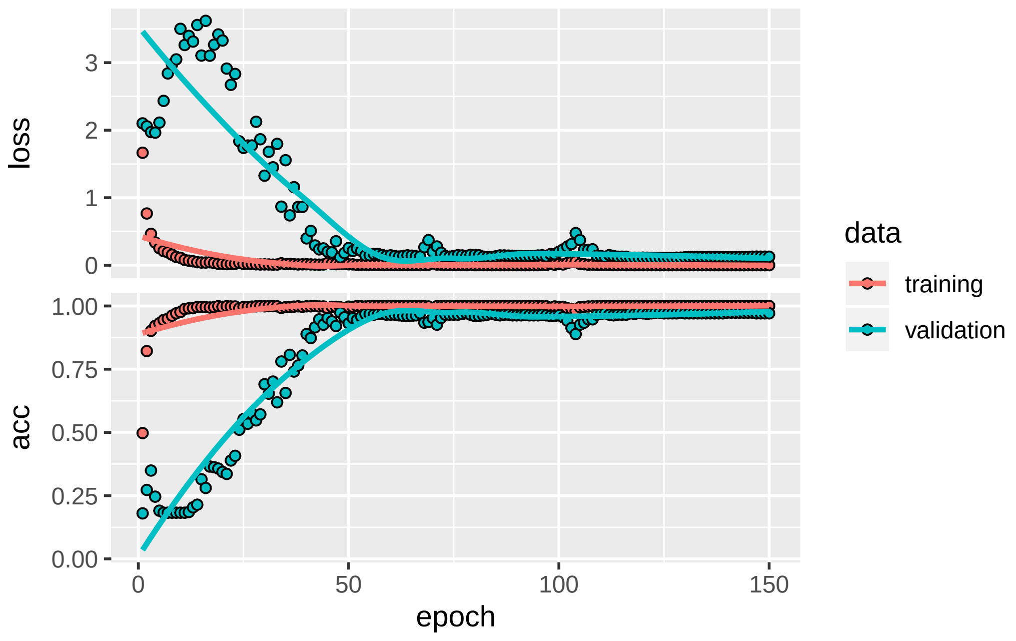

Figure 11: Deep learning classification of a 16 year time series. The location (latitude, longitude) shown at the top of the graph are in geographic coordinate system (WGS84 *datum*).

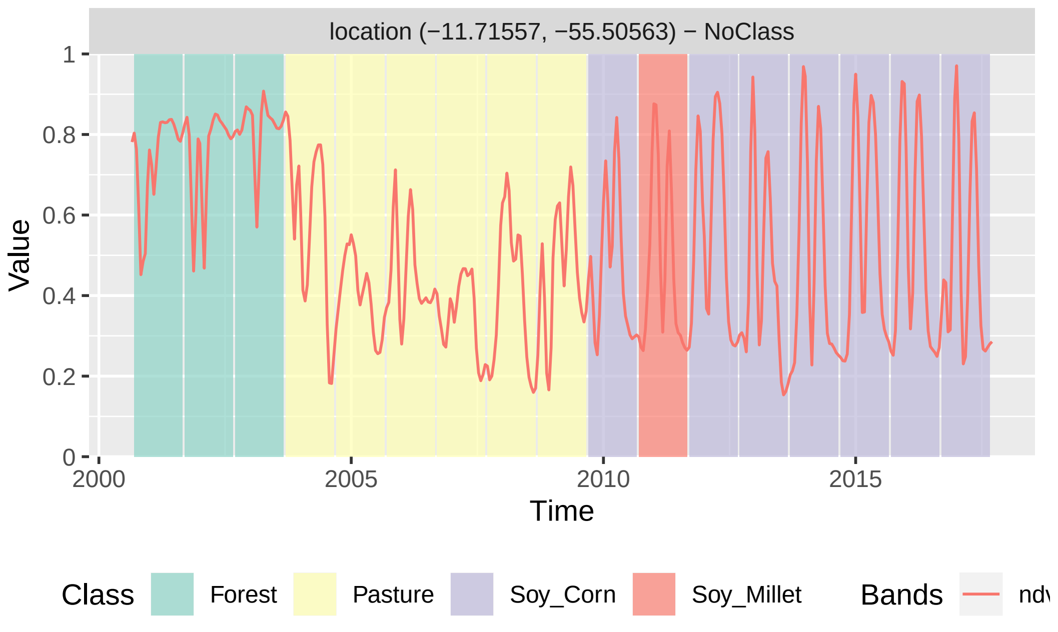

Figure 12: Deep learning classification of a 16 year time series. The location (latitude, longitude) shown at the top of the graph are in geographic coordinate system (WGS84 *datum*).

## References

Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

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

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

Bradley Efron and Trevor Hastie. *Computer age statistical inference*, volume 5. Cambridge University Press, 2016.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

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

Victor Maus, Gilberto Camara, Ricardo Cartaxo, Alber Sanchez, Fernando M Ramos, and Gilberto R de Queiroz. A Time-Weighted Dynamic Time Warping method for Land-Use and Land-Cover mapping. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(8):3729 – 3739, 2016.

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

Aaron E. Maxwell, Timothy A. Warner, and Fang Fang. Implementation of machine-learning classification in remote sensing: an applied review. *International Journal of Remote Sensing*, 39(9):2784–2817, 2018. doi: 10.1080/01431161.2018.1433343.

G. Mountrakis, J. Im, and C. Ogole. Support vector machines in remote sensing: A review. *ISPRS Journal of Photogrammetry and Remote Sensing*, 66(3):247–259, 2011.

Charlotte Pelletier, Geoffrey I Webb, and François Petitjean. Temporal convolutional neural network for the classification of satellite image time series. *Remote Sensing*, 11 (5):523, 2019. https://www.mdpi.com/2072-4292/11/5/523.

Michelle Cristina Araujo Picoli, Gilberto Camara, Ieda Sanches, Rolf Simões, Alexandre Carvalho, Adeline Maciel, Alexandre Coutinho, Julio Esquerdo, João Antunes, Rodrigo Anzolin Begotti, et al. Big Earth observation time series analysis for monitoring brazilian agriculture. *ISPRS Journal of Photogrammetry and Remote Sensing*, 145: 328–339, 2018.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Marvin Wright and Andreas Ziegler. ranger: A fast implementation of random forests for high dimensional data in c++ and r. *Journal of Statistical Software*, 77(1):1–17, 2017. ISSN 1548-7660. doi: 10.18637/jss.v077.i01. URL https://www.jstatsoft.org/v077/i01.