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.

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`)

Data used in the machine learning examples

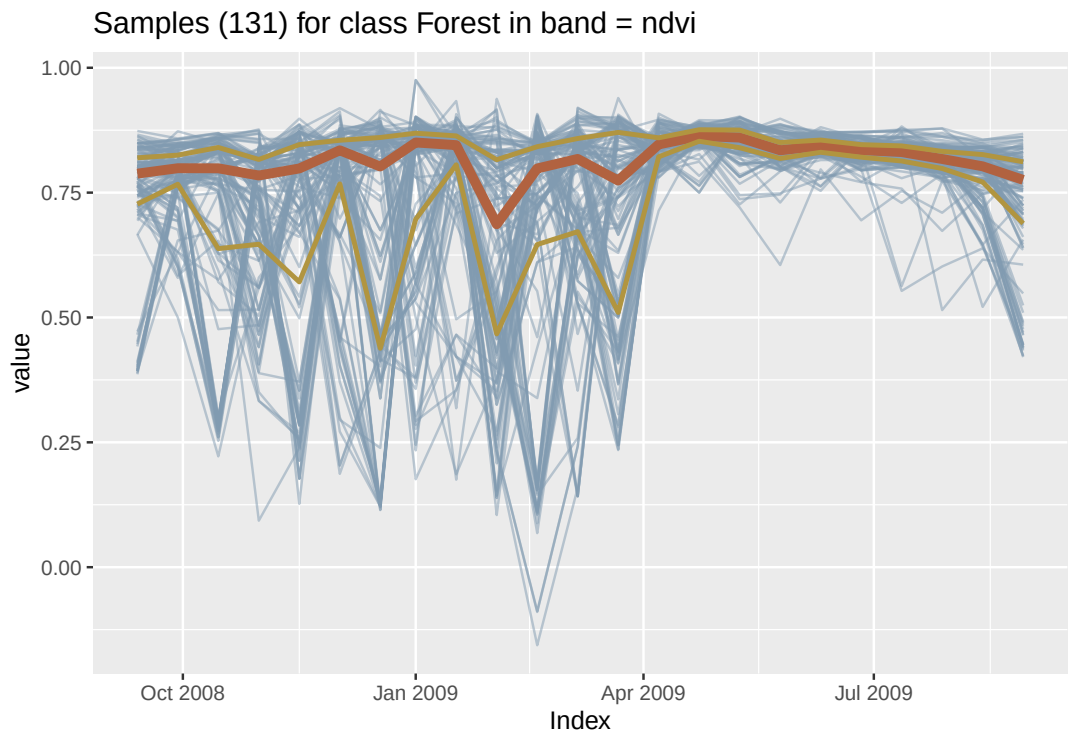
For the machine learning examples, we use a data set containing a sits tibble with time series samples from Brazilian Mato Grosso State (Amazon and Cerrado biomes). The samples are from many sources. It has 9 classes ("Cerrado", "Fallow_Cotton", "Forest", "Millet_Cotton", "Pasture", "Soy_Corn", "Soy_Cotton", "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 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.

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.

```
# Select a subset of the samples to be plotted
# Retrieve the set of samples for the Mato Grosso region
mt_2bands  <- sits_select_bands(samples_mt_4bands, ndvi, evi)
mt_2bands %>%
    sits_select_bands(ndvi) %>%
```

```
    dplyr::filter(label == "Forest") %>%
    plot()
```



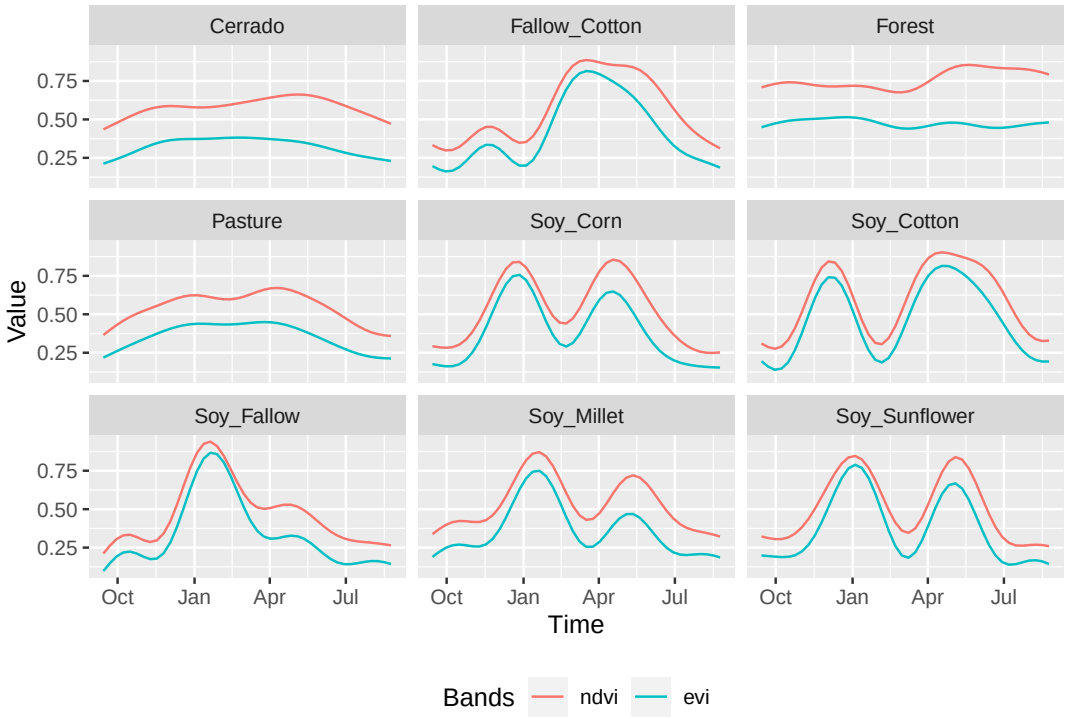
In the above plot, the thick red line is the median value for each time instance and the yellow lines are the first and third interquartile ranges. Visually, one can see that samples labelled as "Forest" are distinguishable from those of "Cerrado" and "Pasture"; in turn, these latter classes have many 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
mt_2bands %>%
    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.

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`.

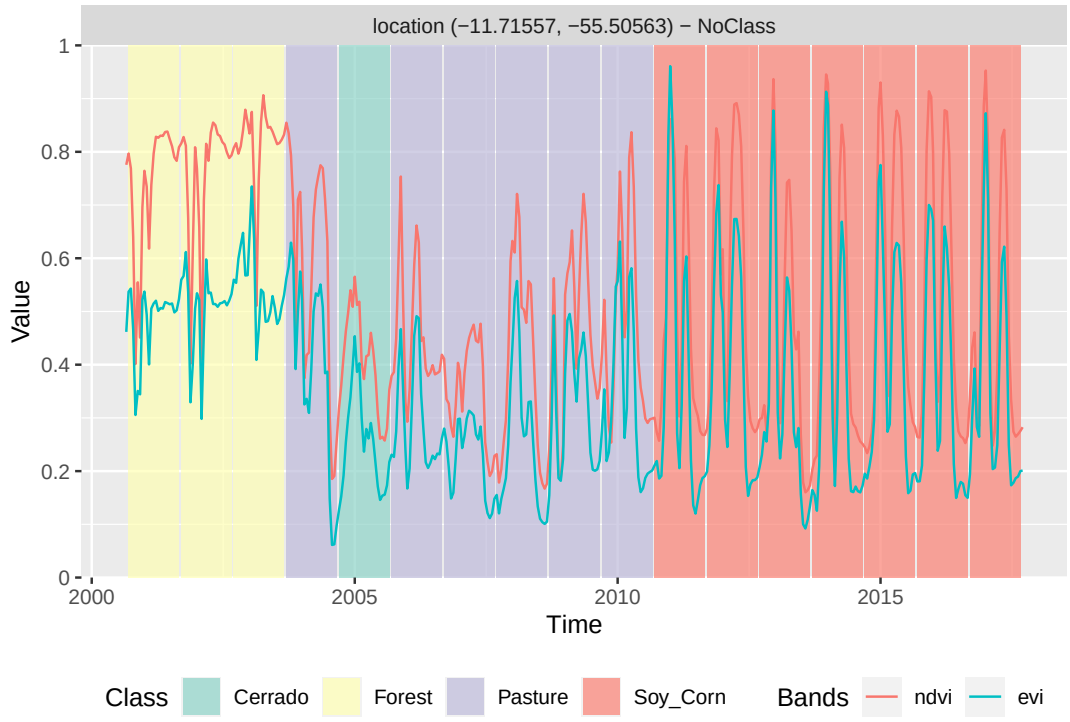
Random forests

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(mt_2bands, sits_rfor(num_trees = 500))
# Classify using Random Forest model and plot the result
point_mt_2bands <- sits_select_bands(point_mt_6bands, ndvi, evi)
class.tb <- point_mt_2bands %>%
    sits_whittaker(lambda = 0.2, bands_suffix = "") %>%
    sits_classify(rfor_model) %>%
    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 Cerrado
##  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 Pasture
## 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 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.

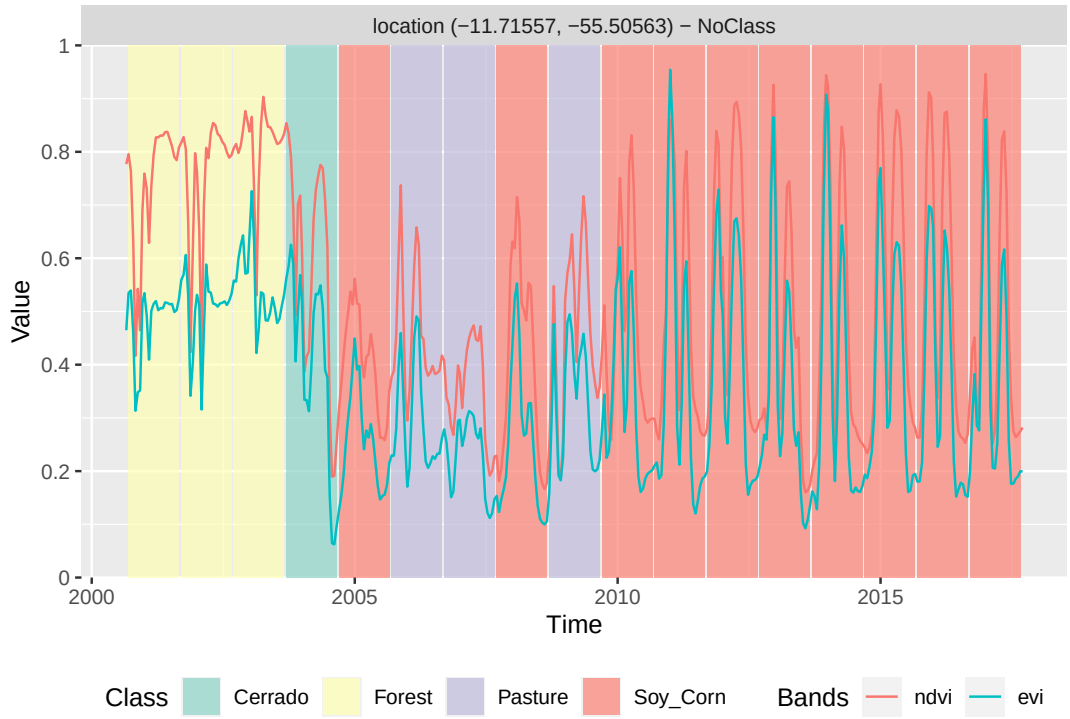
Support Vector Machines

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(mt_2bands,
                        ml_method = sits_svm(kernel = "radial",
                                             cost = 100))
# Classify using SVM model and plot the result
class.tb <- point_mt_2bands %>%
    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.

Extreme Gradient Boosting

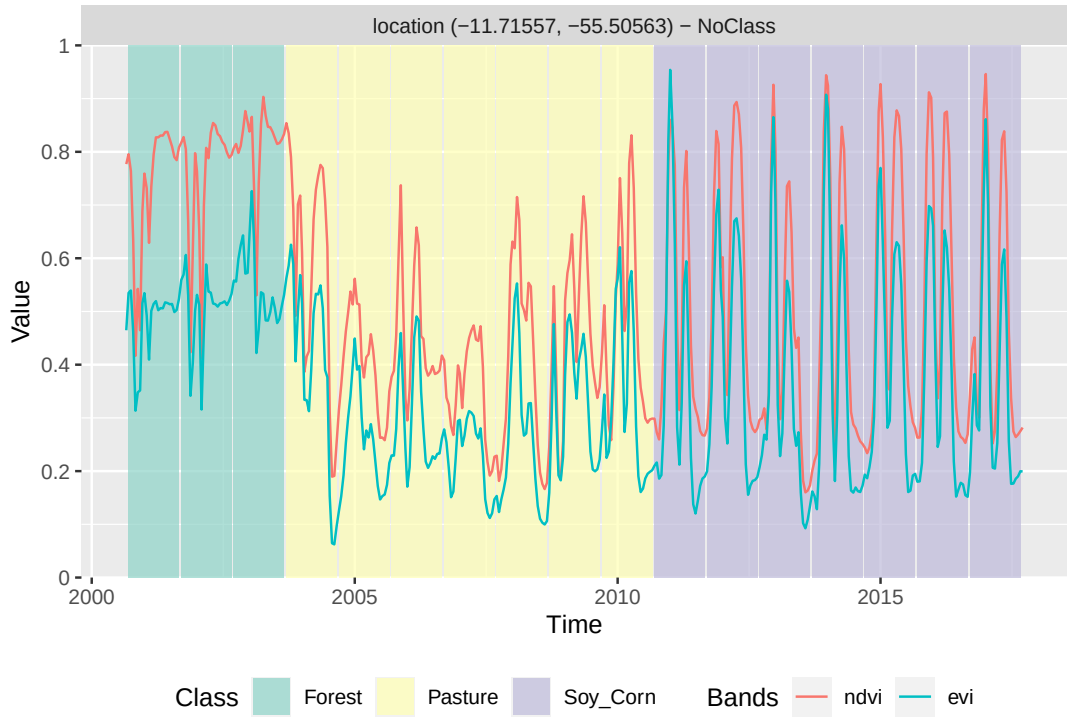
Boosting techniques are based on the idea of starting from a weak predictor and then improving performance sequentially by fitting better model at each iteration. It starts by fitting a simple classifier to the training data. Then it uses the residuals of the regression to build a better prediction. 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(mt_2bands, sits_xgboost(eta = 0.3,
                                                gamma = 0.01,
                                                max_depth = 6,
                                                subsample = 0.9,
                                                nrounds = 100,
                                                verbose = FALSE))
# Classify using SVM model and plot the result
class.tb <- point_mt_2bands %>%
    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.

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 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.

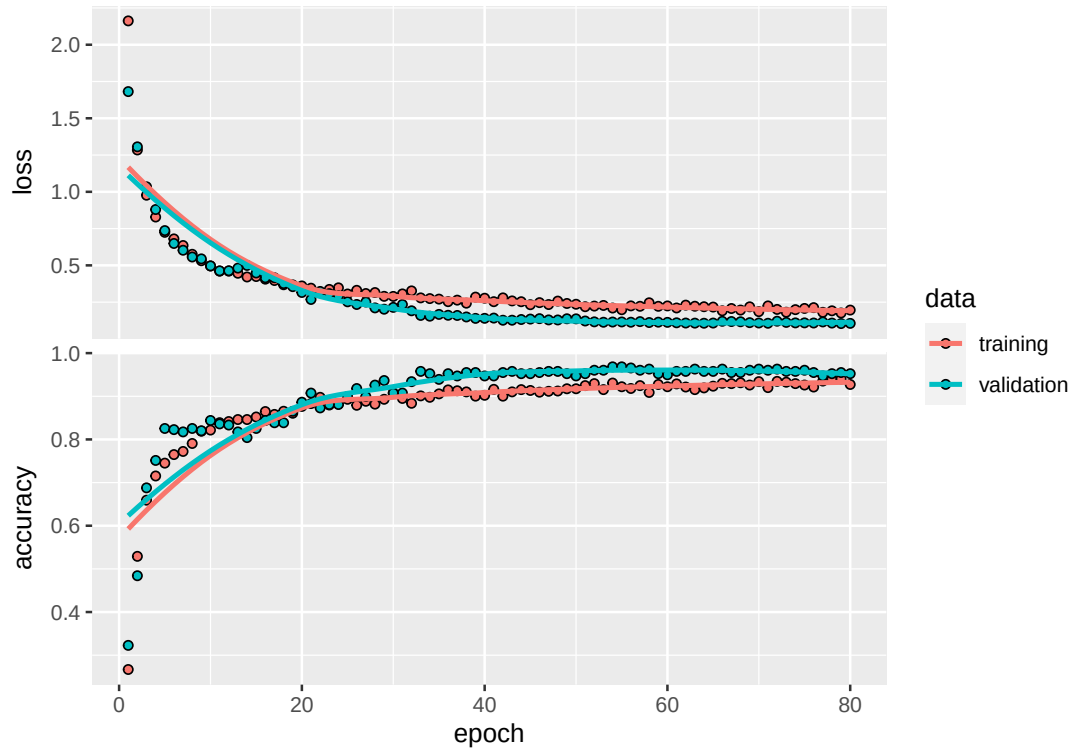
The dropout rates have a huge impact on the performance of MLP classifiers. Dropout is a technique for randomly dropping units from the neural network during training [Srivastava et al., 2014]. By randomly discarding some neurons, dropout reduces overfitting. It is a counter-intuitive idea that works 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 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 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(mt_2bands, 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,
```

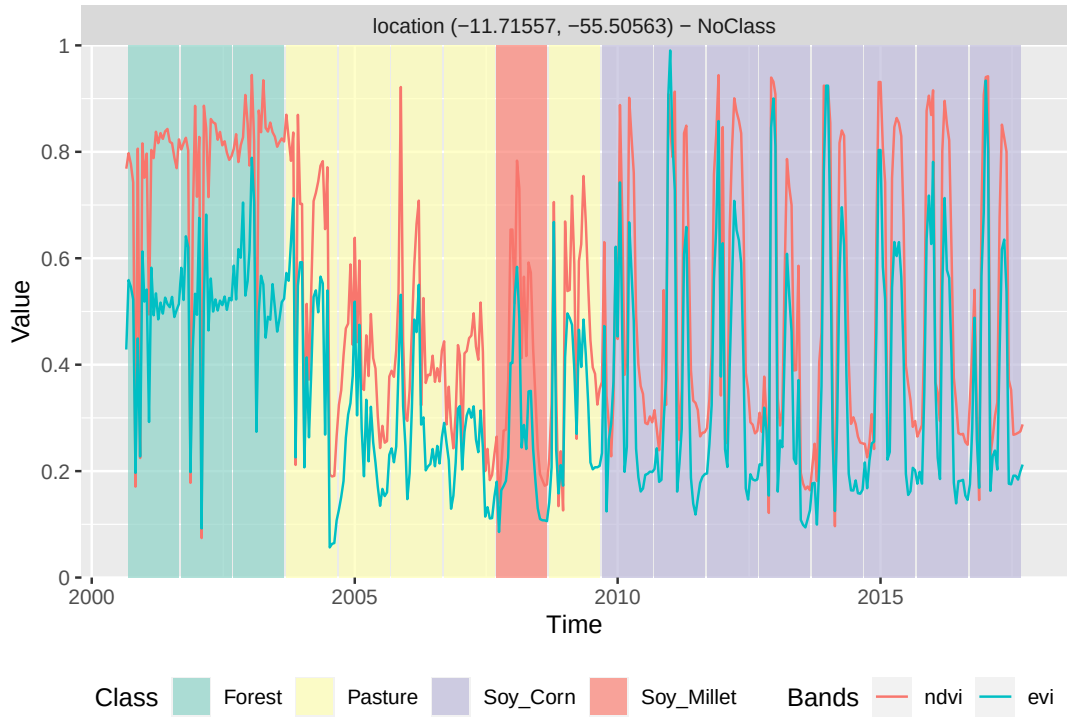
```
                        validation_split = 0.2) )

# 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
class.tb <- point_mt_2bands %>%
    sits_classify(mlp_model) %>%
    plot(bands = c("ndvi", "evi"))
```



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 et al., 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 et al. [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).

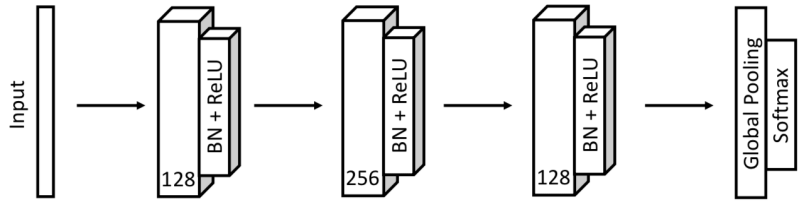
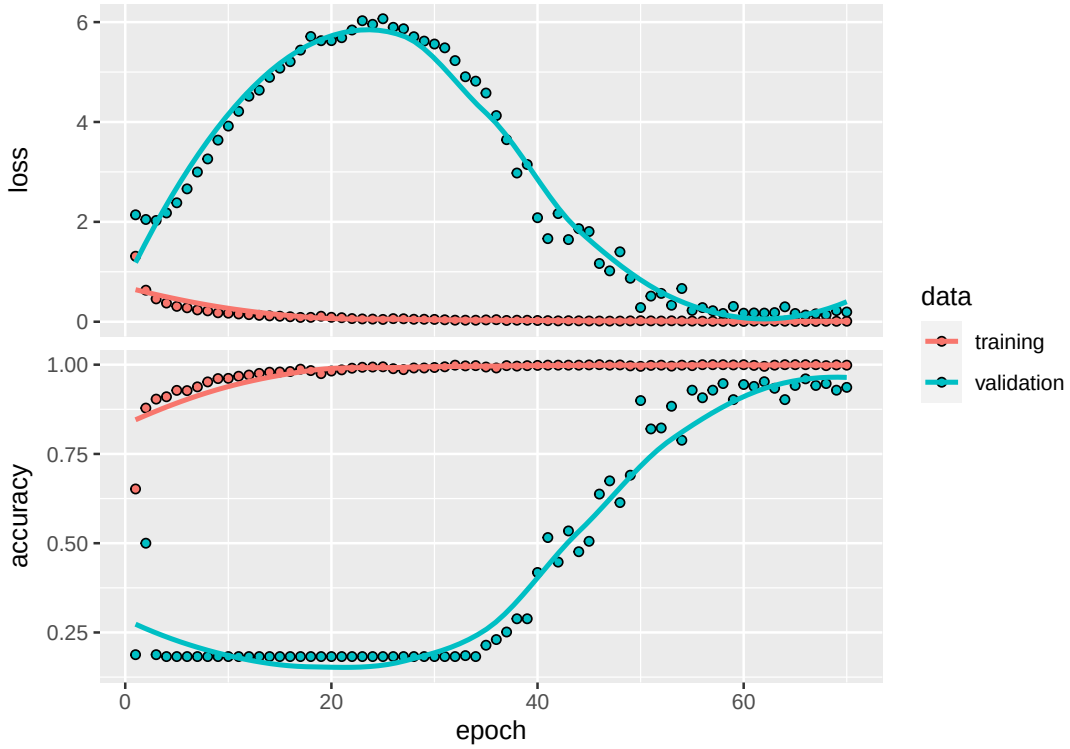


Figure 1: Structure of fullCNN architecture (source: Wang et al.(2017))

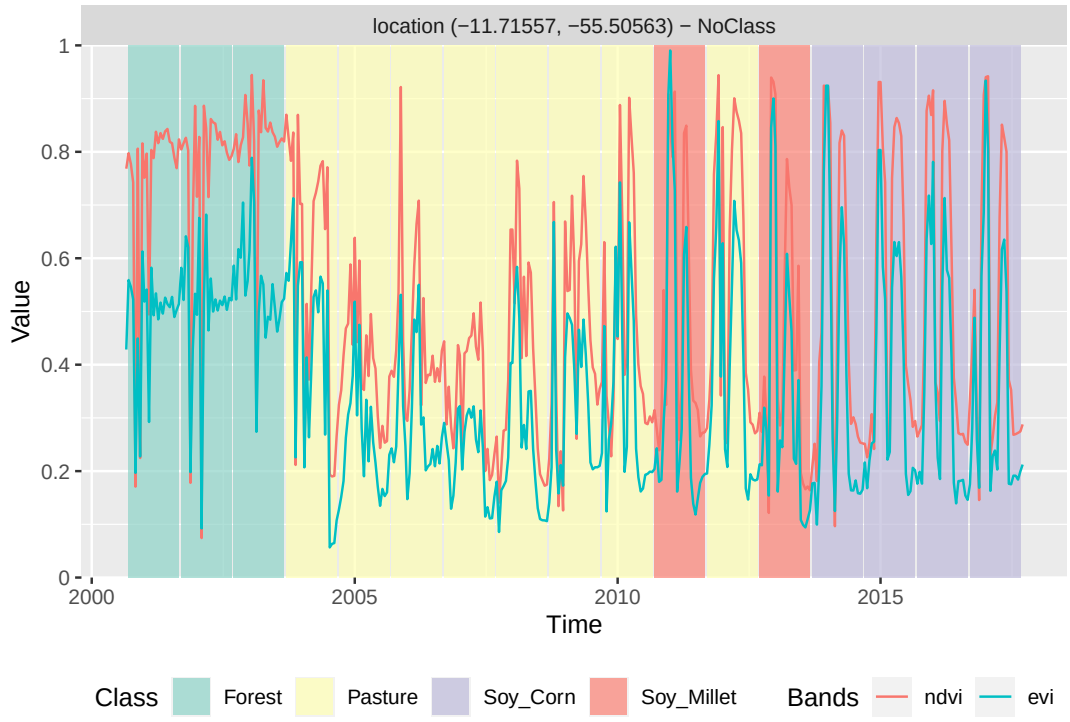
The `sits_FCN` function uses the architecture proposed by Wang as its default, and allows the users to experiment with different settings. The `layers` parameter controls the number of layers and the number of filters in each layer. The `kernels` parameters controls the size of the convolution kernels 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(mt_2bands,
                        sits_FCN(
                            layers           = c(64, 128, 64),
                            kernels          = c(8, 5, 3),
                            activation       = 'relu',
                            L2_rate          = 1e-06,
                            epochs           = 70,
                            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
class.tb <- point_mt_2bands %>%
    sits_classify(fcn_model) %>%
    plot(bands = c("ndvi", "evi"))
```



Residual 1D CNN Networks (ResNet)

The Residual Network (ResNet) is a variation of the fullCNN network proposed by Wang et al. [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].

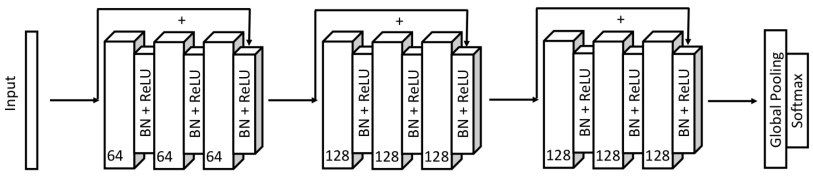
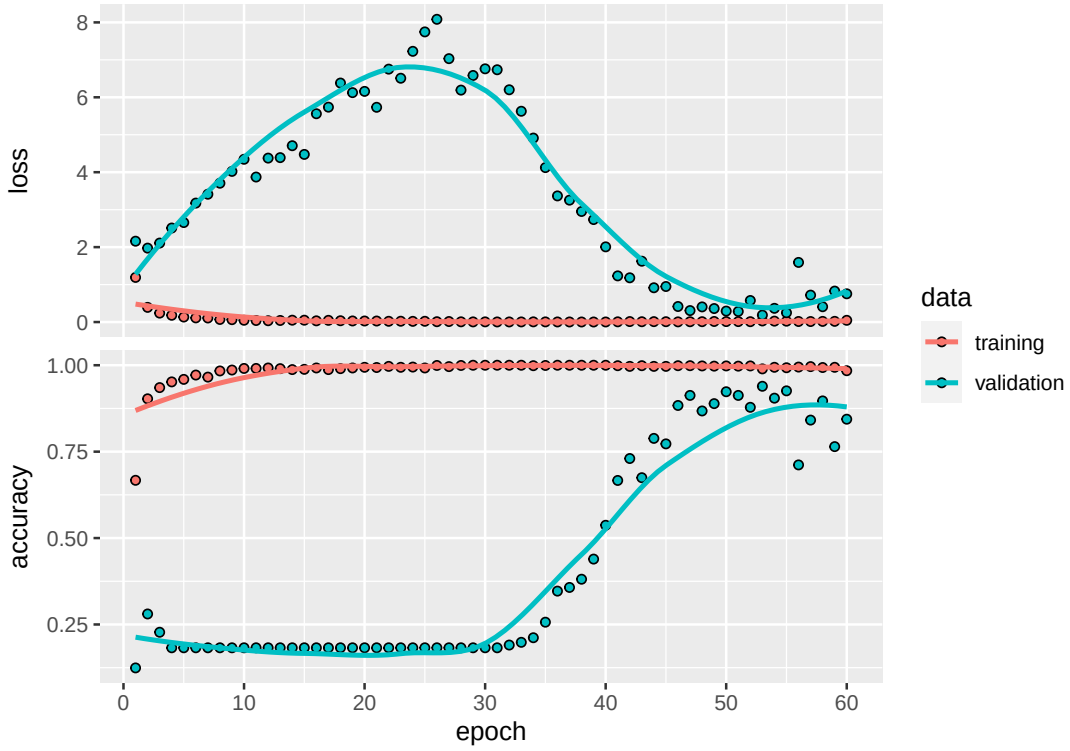


Figure 2: Structure of ResNet architecture (source: Wang et al.(2017))

In SITS, the ResNet is implemented using the `sits_resnet` function. The default parameters are those proposed by Wang et al. [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

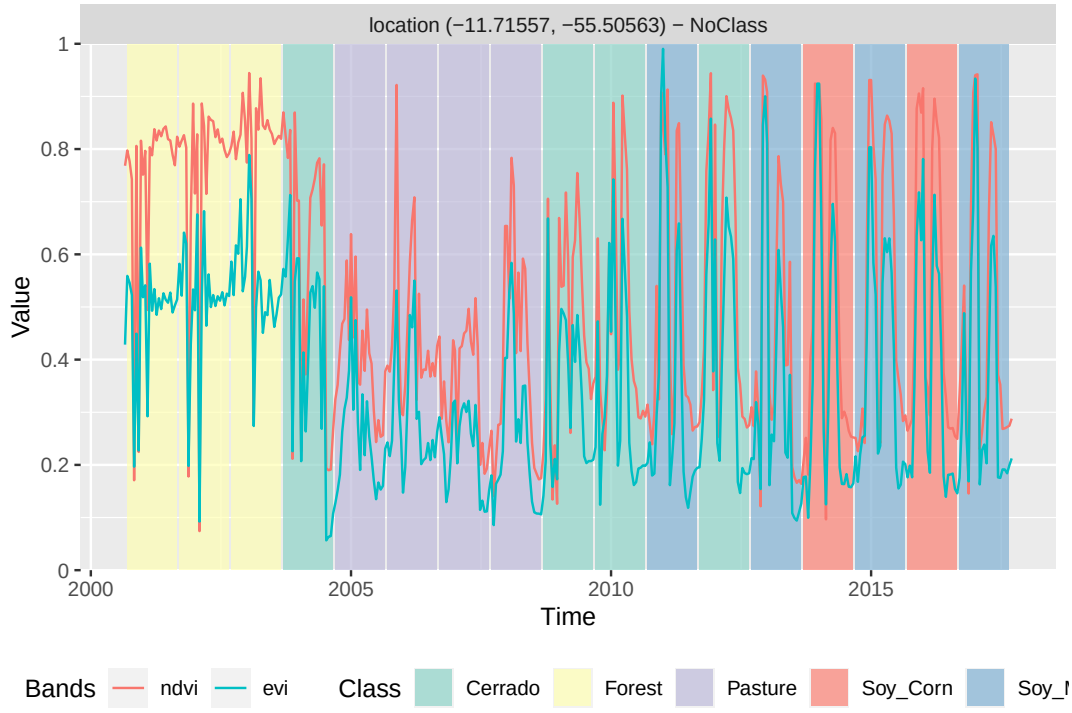
block. By default, the model implements three blocks, the first with 64 filters and the others with 128 filters. Users can control the number of blocks and filter size by changing this parameter. The parameter `kernels` controls the size the of kernels of the three layers inside each block. Wang et al. [2017] proposes to use kernels of sizes 8,5, and 3. 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 et al. [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
resnet_model <- sits_train(mt_2bands,
                       sits_ResNet(
                          blocks              = c(32, 64, 64),
                          kernels             = c(9, 7, 3),
                          activation          = 'relu',
                          epochs              = 60,
                          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
class.tb <- point_mt_2bands %>%
    sits_classify(resnet_model) %>%
    plot(bands = c("ndvi", "evi"))
```



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 et al. [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 et al., 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.

The function `sits_tempCNN` implements the model, using the default parameters proposed by Pelletier et al. [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

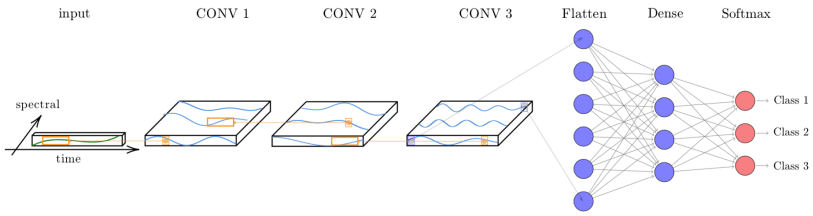
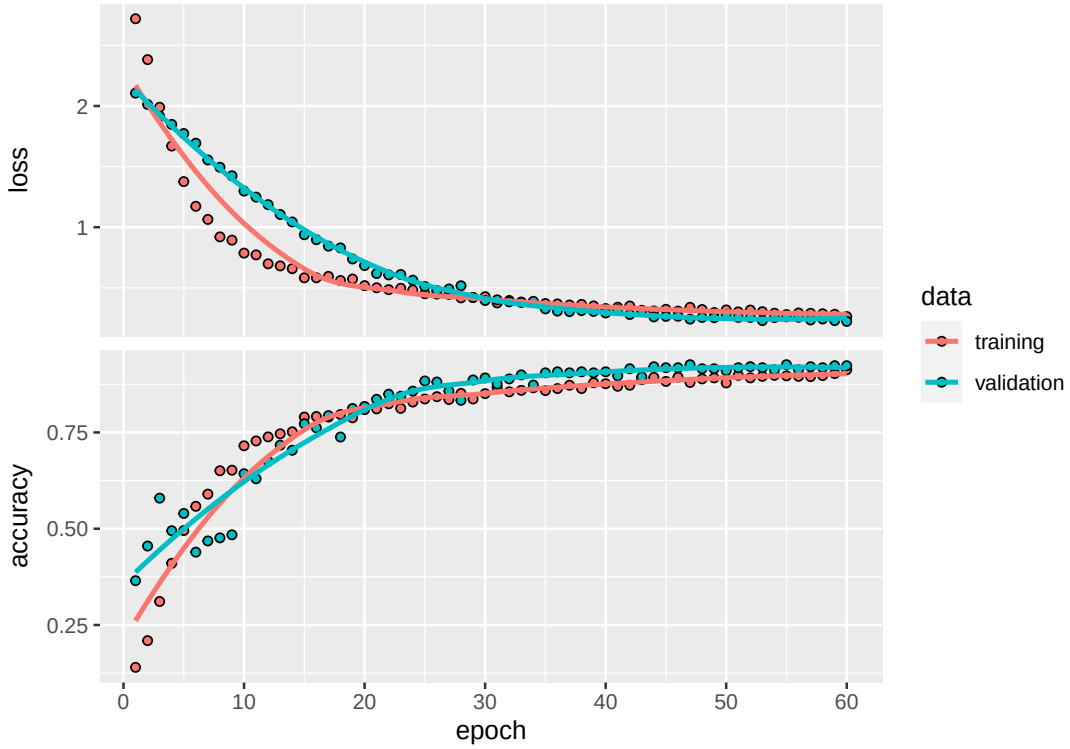


Figure 3: Structure of tempCNN architecture (source: Pelletier et al.(2019))

`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%.

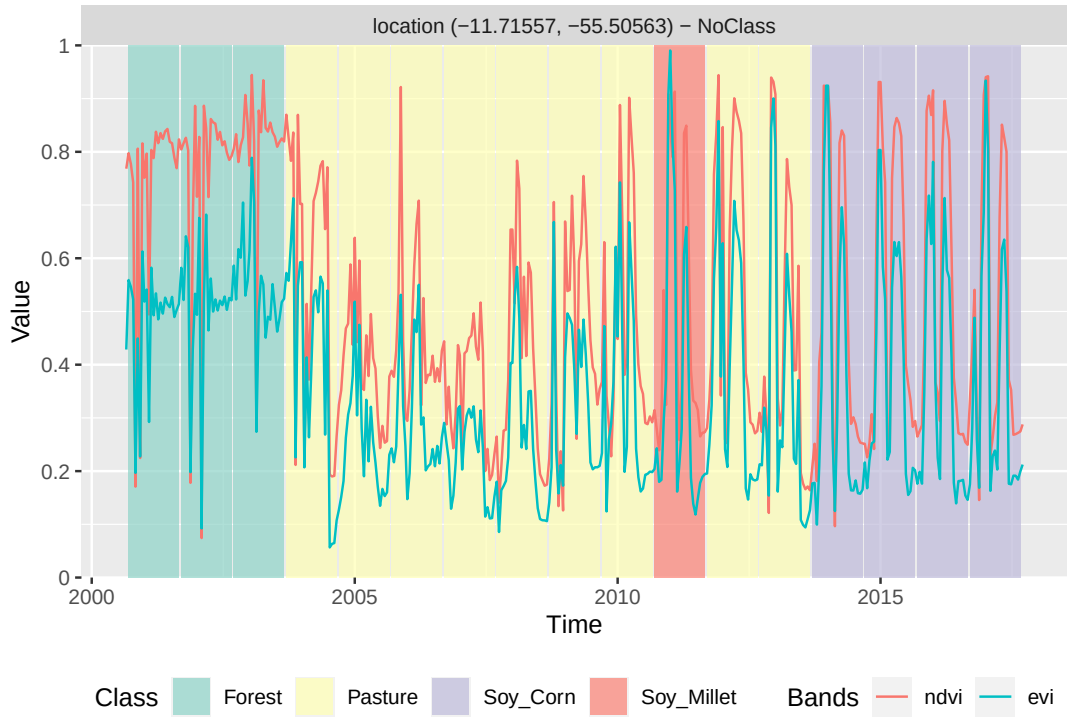
```
# train a machine learning model using tempCNN
tCNN_model <- sits_train(mt_2bands,
                         sits_TempCNN(
                            cnn_layers           = c(32, 32, 32),
                            cnn_kernels          = c(5, 5, 5),
                            cnn_activation       = '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
class.tb <- point_mt_2bands %>%
    sits_classify(tCNN_model) %>%
    plot(bands = c("ndvi", "evi"))
```



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.

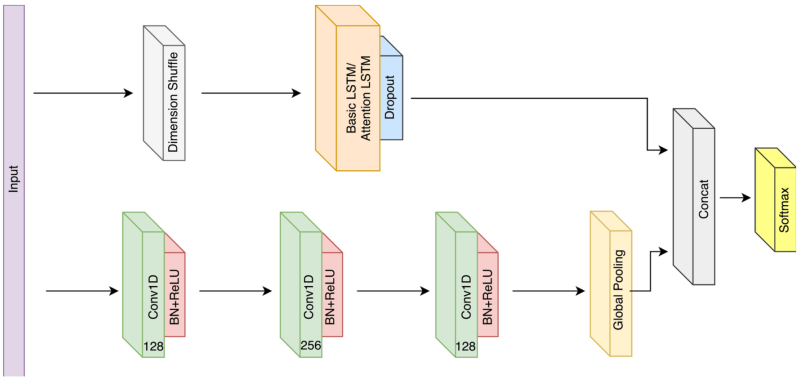
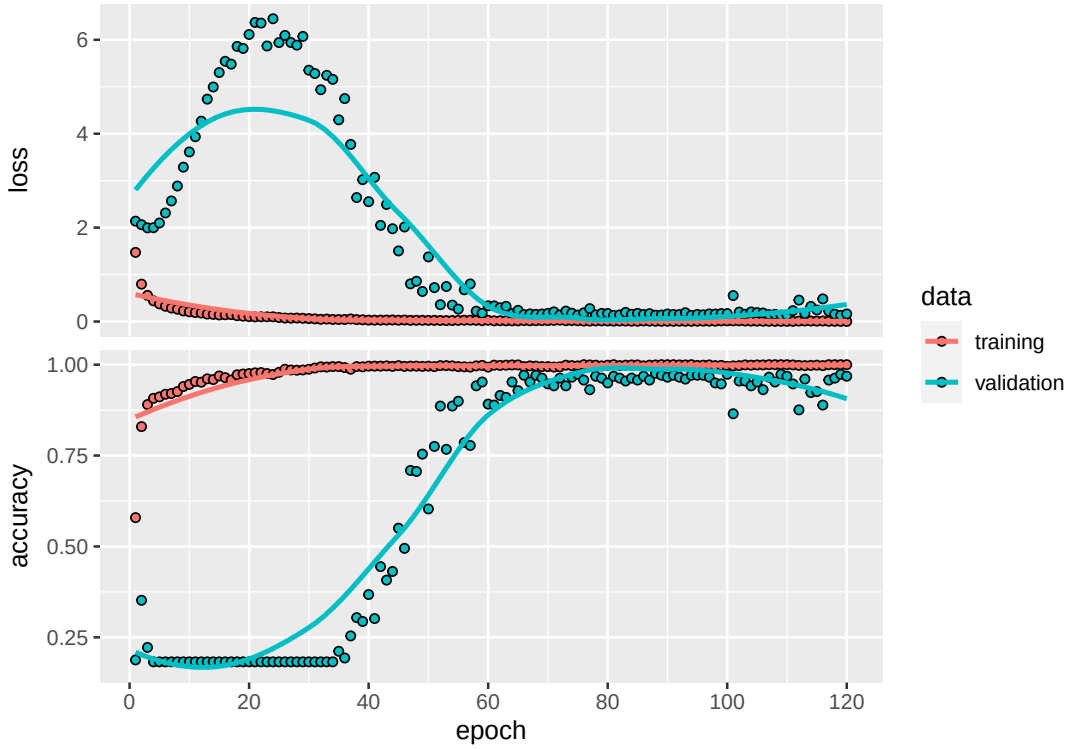


Figure 4: LSTM Fully Convolutional Networks for Time Series Classification (source: Karim et al.(2019))

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}, controled by the parameters `cnn_layers` and `cnn_kernels`. One should experiment with these parameters, and consider the simulations carried out by Pelletier et al. [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%.

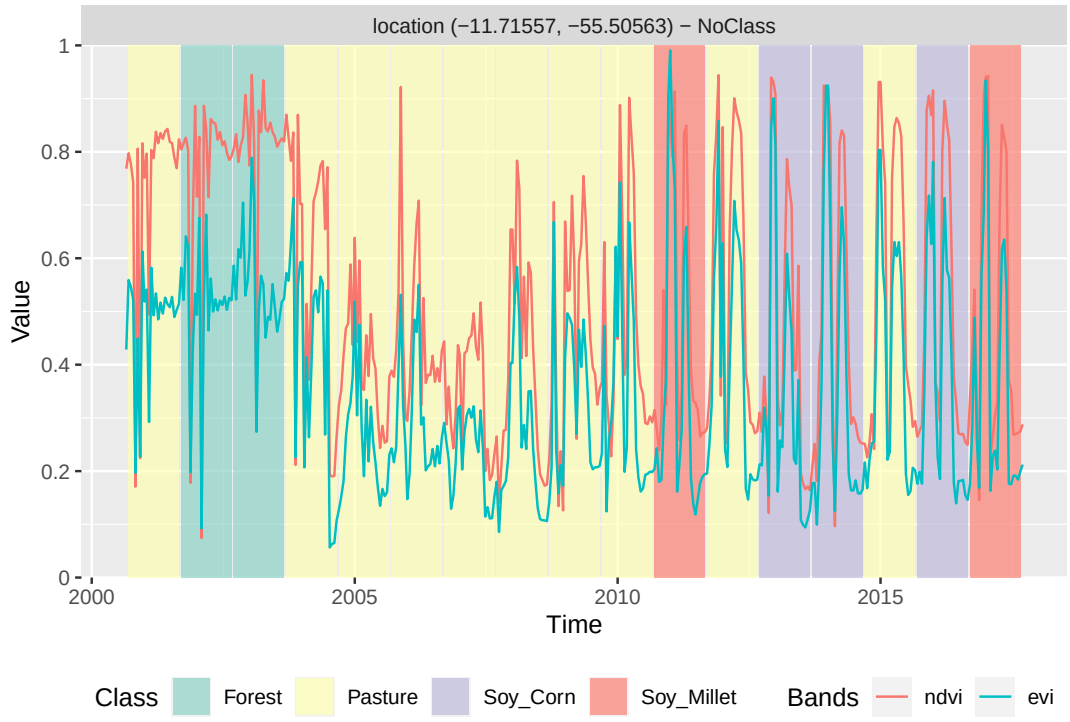
```
lstm_fcn_model <- sits_train(mt_2bands,
                        sits_LSTM_FCN(
                           lstm_units          = 16,
                           lstm_dropout        = 0.80,
                           cnn_layers          = c(64, 64, 64),
                           cnn_kernels         = 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
class.tb <- point_mt_2bands %>%
    sits_classify(lstm_fcn_model) %>%
    plot(bands = c("ndvi", "evi"))
```



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 et al., 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 et al. [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 et al., 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())
# prints the output confusion matrix and statistics
sits_conf_matrix(prediction.mx)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Cerrado Pasture
##    Cerrado     396      12
##    Pasture       4     334
##
##           Accuracy : 0.9786
##             95% CI : (0.9654, 0.9877)
##
##              Kappa : 0.9568
##
##  Prod Acc  Cerrado : 0.9900
##  Prod Acc  Pasture : 0.9653
##  User Acc  Cerrado : 0.9706
##  User Acc  Pasture : 0.9882
##
```

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_mt_4bands")

# create a list to store the results
results <- list()

# adjust the multicores parameters to suit your machine
```

```r
## SVM model
conf_svm.tb <- sits_kfold_validate(samples_mt_4bands,
                                   folds = 5,
                                   multicores = 2,
                                ml_method = sits_svm(kernel = "radial", cost = 10))

print("== Confusion Matrix = SVM ======================")
```

```
## [1] "== Confusion Matrix = SVM ======================"
```

```r
conf_svm.mx <- sits_conf_matrix(conf_svm.tb)
```

```
## Confusion Matrix and Statistics
##
##                Reference
## Prediction      Pasture Soy_Corn Soy_Millet Soy_Cotton Fallow_Cotton
##   Pasture           318        3          6          2              4
##   Soy_Corn            5      345         12          8              0
##   Soy_Millet          4       11        158          0              0
##   Soy_Cotton          3        2          1        335              5
##   Fallow_Cotton       1        0          0          3             19
##   Soy_Sunflower       0        0          1          0              0
##   Cerrado            13        3          2          3              1
##   Forest              0        0          0          0              0
##   Soy_Fallow          0        0          0          1              0
##                Reference
## Prediction      Soy_Sunflower Cerrado Forest Soy_Fallow
##   Pasture                   0       4      2          0
##   Soy_Corn                  9       0      0          0
##   Soy_Millet                1       0      0          0
##   Soy_Cotton                0       0      0          1
##   Fallow_Cotton             0       0      0          0
##   Soy_Sunflower            16       0      0          0
##   Cerrado                   0     373      6          0
##   Forest                    0       2    123          0
##   Soy_Fallow                0       0      0         86
##
## Overall Statistics
##
##  Accuracy : 0.9371
##    95% CI : (0.9252, 0.9476)
##
##     Kappa : 0.9248
##
## Statistics by Class:
##
```

```
##                         Class: Pasture Class: Soy_Corn Class: Soy_Millet
## Prod Acc (Sensitivity)          0.9244         0.9478           0.8778
## Specificity                     0.9864         0.9777           0.9907
## User Acc (Pos Pred Value)       0.9381         0.9103           0.9080
## Neg Pred Value                  0.9833         0.9874           0.9872
##                           Class: Soy_Cotton Class: Fallow_Cotton
## Prod Acc (Sensitivity)               0.9517               0.6552
## Specificity                          0.9922               0.9979
## User Acc (Pos Pred Value)            0.9654               0.8261
## Neg Pred Value                       0.9890               0.9946
##                         Class: Soy_Sunflower Class: Cerrado Class: Forest
## Prod Acc (Sensitivity)               0.6154        0.9842       0.9389
## Specificity                          0.9995        0.9815       0.9989
## User Acc (Pos Pred Value)            0.9412        0.9302       0.9840
## Neg Pred Value                       0.9947        0.9960       0.9955
##                           Class: Soy_Fallow
## Prod Acc (Sensitivity)               0.9885
## Specificity                          0.9994
## User Acc (Pos Pred Value)            0.9885
## Neg Pred Value                       0.9994
```

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

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


# ============== Random Forest ==============================

conf_rfor.tb <- sits_kfold_validate(samples_mt_4bands,
                                    folds = 5,
                                    multicores = 1,
                                    ml_method = sits_rfor(num_trees = 500))
print("== Confusion Matrix = RFOR ======================")
```

```
## [1] "== Confusion Matrix = RFOR ======================"
```

```
conf_rfor.mx <- sits_conf_matrix(conf_rfor.tb)
```

```
## Confusion Matrix and Statistics
##
##                Reference
## Prediction      Pasture Soy_Corn Soy_Millet Soy_Cotton Fallow_Cotton
##   Pasture           340        2          4          1             2
##   Soy_Corn            1      351          7         13             1
```

```
##   Soy_Millet          0        9        166          0             0
##   Soy_Cotton          1        2          2        338             3
##   Fallow_Cotton       0        0          0          0            23
##   Soy_Sunflower       0        0          0          0             0
##   Cerrado             2        0          0          0             0
##   Forest              0        0          0          0             0
##   Soy_Fallow          0        0          1          0             0
##                 Reference
## Prediction      Soy_Sunflower Cerrado Forest Soy_Fallow
##   Pasture                   0       0      1          0
##   Soy_Corn                 10       0      0          0
##   Soy_Millet                1       0      0          1
##   Soy_Cotton                0       0      0          0
##   Fallow_Cotton             0       0      0          0
##   Soy_Sunflower            15       0      0          0
##   Cerrado                   0     379      0          0
##   Forest                    0       0    130          0
##   Soy_Fallow                0       0      0         86
##
## Overall Statistics
##
##  Accuracy : 0.9662
##    95% CI : (0.957, 0.9739)
##
##     Kappa : 0.9596
##
## Statistics by Class:
##
##                       Class: Pasture Class: Soy_Corn Class: Soy_Millet
## Prod Acc (Sensitivity)         0.9884         0.9643           0.9222
## Specificity                    0.9935         0.9791           0.9936
## User Acc (Pos Pred Value)      0.9714         0.9164           0.9379
## Neg Pred Value                 0.9974         0.9914           0.9918
##                          Class: Soy_Cotton Class: Fallow_Cotton
## Prod Acc (Sensitivity)              0.9602               0.7931
## Specificity                         0.9948               1.0000
## User Acc (Pos Pred Value)           0.9769               1.0000
## Neg Pred Value                      0.9909               0.9968
##                       Class: Soy_Sunflower Class: Cerrado Class: Forest
## Prod Acc (Sensitivity)               0.5769        1.0000       0.9924
## Specificity                          1.0000        0.9987       1.0000
## User Acc (Pos Pred Value)            1.0000        0.9948       1.0000
## Neg Pred Value                       0.9941        1.0000       0.9994
##                          Class: Soy_Fallow
## Prod Acc (Sensitivity)              0.9885
## Specificity                         0.9994
## User Acc (Pos Pred Value)           0.9885
```

```
## Neg Pred Value                         0.9994
```

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

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

# ============== XGBOOST ==============================
# extreme gradient boosting
conf_xgb.tb <- sits_kfold_validate(samples_mt_4bands,
                                   folds = 5,
                                   multicores = 32,
                                   ml_method = sits_xgboost(nthread = 32))

# print the accuracy of the Multinomial log-linear
print("== Confusion Matrix = XGB ======================")
```

```
## [1] "== Confusion Matrix = XGB ======================"
```

```
conf_xgb.mx <- sits_conf_matrix(conf_xgb.tb)
```

```
## Confusion Matrix and Statistics
##
##                Reference
## Prediction      Pasture Soy_Corn Soy_Millet Soy_Cotton Fallow_Cotton
##   Pasture           328        2         10          1              4
##   Soy_Corn            2      339          9         17              0
##   Soy_Millet          1       17        156          0              0
##   Soy_Cotton          2        3          1        333              5
##   Fallow_Cotton       0        0          1          1             19
##   Soy_Sunflower       1        2          0          0              0
##   Cerrado            10        1          2          0              0
##   Forest              0        0          0          0              1
##   Soy_Fallow          0        0          1          0              0
##                Reference
## Prediction      Soy_Sunflower Cerrado Forest Soy_Fallow
##   Pasture                   1       4      0          0
##   Soy_Corn                  9       0      0          1
##   Soy_Millet                3       0      1          2
##   Soy_Cotton                1       0      0          0
##   Fallow_Cotton             0       0      0          0
##   Soy_Sunflower            12       0      0          0
##   Cerrado                   0     374      0          0
##   Forest                    0       1    130          0
##   Soy_Fallow                0       0      0         84
```

```
##
## Overall Statistics
##
##  Accuracy : 0.9382
##    95% CI : (0.9263, 0.9486)
##
##     Kappa : 0.9261
##
## Statistics by Class:
##
##                        Class: Pasture Class: Soy_Corn Class: Soy_Millet
## Prod Acc (Sensitivity)          0.9535         0.9313           0.8667
## Specificity                     0.9858         0.9751           0.9860
## User Acc (Pos Pred Value)       0.9371         0.8992           0.8667
## Neg Pred Value                  0.9896         0.9835           0.9860
##                           Class: Soy_Cotton Class: Fallow_Cotton
## Prod Acc (Sensitivity)               0.9460               0.6552
## Specificity                          0.9922               0.9989
## User Acc (Pos Pred Value)            0.9652               0.9048
## Neg Pred Value                       0.9877               0.9947
##                         Class: Soy_Sunflower Class: Cerrado Class: Forest
## Prod Acc (Sensitivity)                0.4615        0.9868       0.9924
## Specificity                           0.9984        0.9914       0.9989
## User Acc (Pos Pred Value)             0.8000        0.9664       0.9848
## Neg Pred Value                        0.9925        0.9967       0.9994
##                           Class: Soy_Fallow
## Prod Acc (Sensitivity)               0.9655
## Specificity                          0.9994
## User Acc (Pos Pred Value)            0.9882
## Neg Pred Value                       0.9983
```

```r
# Give a name to the model
conf_xgb.mx$name <- "xgboost"

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

# choose the output directory
WD = getwd()

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

```
## Saved Excel file ./accuracy_mt_ml.xlsx
```

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.

Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

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.

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. LSTM fully convolutional networks for time series classification. *IEEE Access*, 6:1662–1669, 2018.

Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Samuel Harford. Multivariate LSTM-FCNS for time series classification. *Neural Networks*, 116:237–245, 2019.

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.

Marc Rußwurm and Marco Korner. Temporal vegetation modelling using long short-term memory networks for crop identification from medium-resolution multi-spectral satellite images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 11–19, 2017.

Marc Rußwurm and Marco Körner. Multi-temporal land cover classification with sequential recurrent encoders. *ISPRS International Journal of Geo-Information*, 7(4): 129, 2018.

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.

Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. In *2017 international joint conference on neural networks (IJCNN)*, pages 1578–1585. IEEE, 2017.

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.