Project no. 6524105

ATLAS

Artificial Intelligence Theoretical Foundations for Advanced Spatio-Temporal Modelling of Data and Processes

WP5: Integrated platform

Deliverable D5.2

Model Repository

Program for Development of Projects in the field of Artificial Intelligence
https://ai.ipb.ac.rs/

Report prepared by (alphabetic order):
Dimitrije Maletić (IPB)

Report reviewed internally by:
Andreja Stojić (IPB)

Date: 4/3/2021
Type: Public

Summary

/

Keywords: ROOT, R, Python, TMVA.

1. Introduction

The software which will be used for most demanding tasks, i.e., in batch jobs running, is TMVA – a ROOT-integrated toolkit for multivariate data analysis. ROOT is a software framework for data analysis and data storing and retrieval: a powerful tool to cope with the demanding tasks typical of state-of-the-art scientific data analysis. Among its prominent features there are an advanced graphical user interface, ideal for interactive analysis, and an interpreter for the C++ programming language, used for rapid and efficient prototyping and a persistence mechanism for C++ objects, used to write petabytes of data recorded by the Large Hadron Collider experiments every year. TMVA has a number of different MVA methods implemented along with PyMVA and RMVA interfaces for third-party MVA tools based on Python and R. Those two interfaces were created to make powerful external libraries easily accessible with direct integration into the TMVA workflow. All PyMVA and RMVA methods provide the same plug-and-play mechanisms as other TMVA methods. Besides TMVA, standalone R, Bash, C++, Python, or Docker files (notebook) can be used for simpler, less demanding tasks.

2. Method implementation

TMVA implements classification and regression multivariate analysis (MVA) methods. It provides many benefits including:

- training and testing of different MVA methods on the same data sample, enabling the comparison of different methods, including R and Python implementations, or MVA methods through RMVA and PyMVA interfaces;
- methods for preprocessing of input variable transformation options like normalization, decorrelation, principal component decomposition, “uniformization” and “Gaussianization”;
- variable ranking, a ranking of evaluation results by best signal efficiency and purity, and testing versus training efficiency comparison (overtraining check); etc.

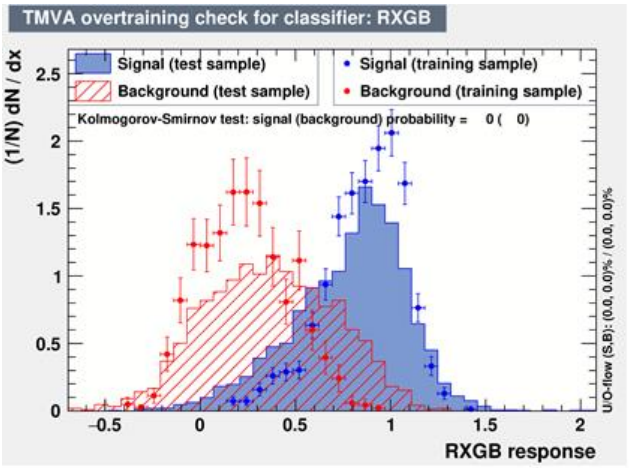
MVA methods implementations that can be selected to be trained, tested, and compared with tunable configurations for each method, are presented in Table 1.

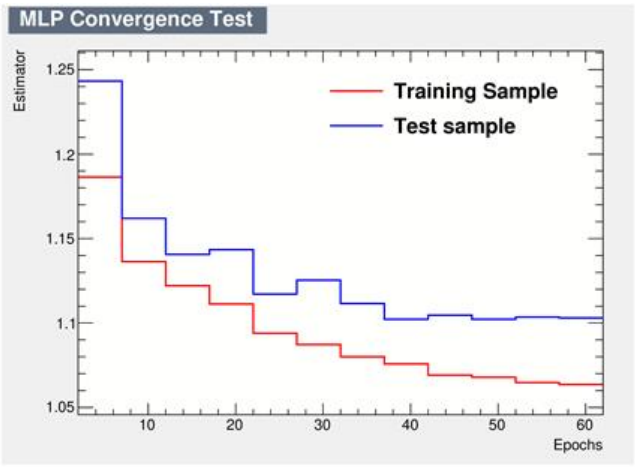
Table 1. ATLAS MVA methods.

Environment	Method
ROOT	Cut optimization
	1-dimensional likelihood ("naive Bayes estimator")
	Mutidimensional likelihood and Nearest-Neighbour methods
	Linear Discriminant Analysis
	Function Discriminant analysis
	Neural Networks (feed-forward Multilayer Perceptrons)
	CUDA-accelerated DNN training
	Multi-core accelerated DNN
	Support Vector Machine
	Boosted Decision Trees
	Friedman's RuleFit (an optimized series of cuts)

R	C50
	RXGB
	RSVM
Python	PyRandomForest
	PyAdaBoost
	PyGTB
	PyXGBoost

PARADOX provides modified installation of ROOT, R, and Python with additional install options and packages that can be accessed by the ATLAS users within the ATLAS platform for batch jobs initiated through the web interface. Figure 1 shows some of the results of classification methods test, with overtaining check for RXGB, MLP convergence test, and the most popular Receiver Operation Characteristic (ROC) plot. ROC shows the comparison of all ATLAS MVA method's performance, including R and Python implementations using RMVA and PyMVA interfaces, ran in parallel on the same training and test samples.





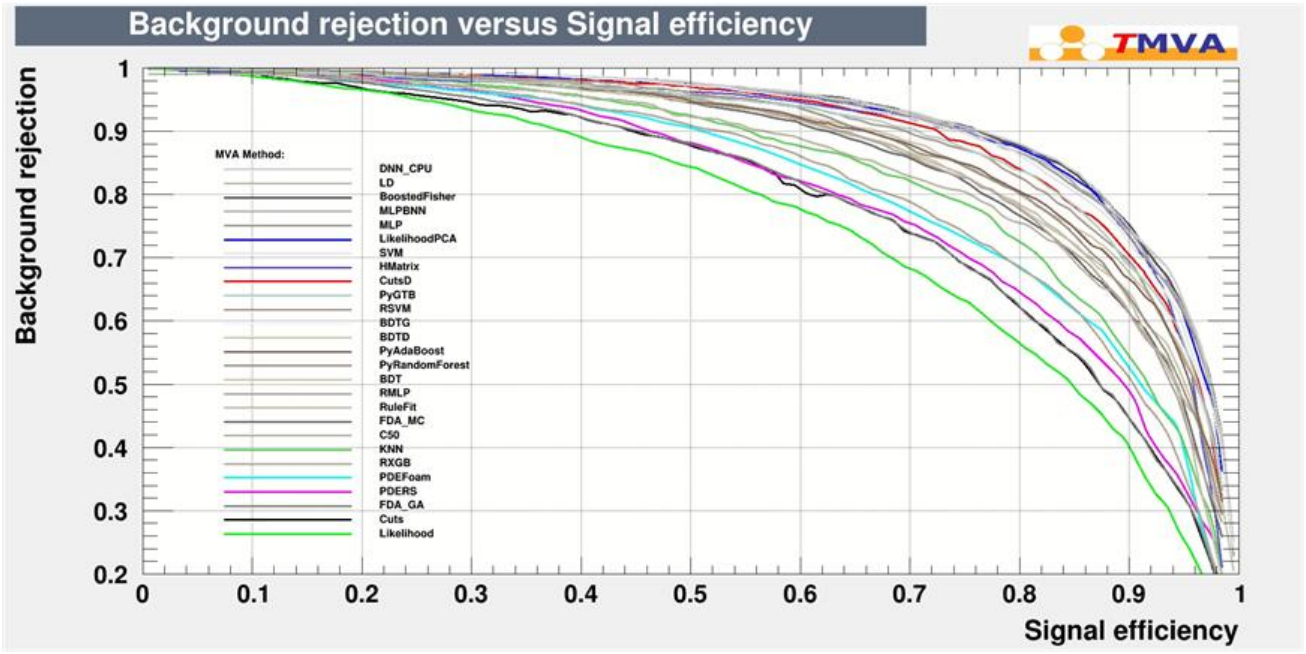


Figure 1: ATLAS MVA method's performance.

At the development stage, users of the ATLAS platform can:

- login to PARADOX gateway using SSH client;
- use bash terminal to develop and test scripts;
- submit jobs for batch processing; and
- retrieve the output using (Win)SCP client.

PARADOX provides basic examples, such as:

- start using TMVA in ROOT;
- batch job submission;
- creating maps using R (*leaflet* example or creating a map of environmental parameter measurement stations);
- check of the SQLite database (DB) access using batch job submission; and
- accessing SQLite DB by the Rscript.

All the information on setting the ATLAS environment and examples, a user gets by sourcing the environment on PARADOX from user's space:

[<user>@paradox ~]$ source /home/dmaletic/source_atlas.sh

The firs time user should get access to R and python local libraries.

Copy (ctrl+shift+c) and paste (ctrl+shift+v) these two lines in terminal:

cd ; mkdir -p R/x86_64-redhat-linux-gnu-library/3.5/; cd R/x86_64-redhat-linux-gnu-library/3.5/; ln -s /home/dmaletic/R/x86_64-redhat-linux-gnu-library/3.5/* . ; cd

cd ; mkdir -p .local/lib/python2.7/site-packages; cd .local/lib/python2.7/site-packages; ln -s /home/dmaletic/.local/lib/python2.7/site-packages/* . ; cd

==========

Setting environment. Setting: module load gnu/4.9.4 python/2.7.6 cmake/3.10.0

==========

You can get initial set of mva examples. Will be updated.

unzip -q /home/dmaletic/tests_atlas.zip

Start using TMVA in ROOT by:

cd tests_atlas/tmva/tmva

root TMVAClassification.C

Also, in directory tests_atlas/tmva you can build binaries by:

make

besides tmva, you can use python methods in pymva or R methods in rmva

cd ~/tests_atlas/R/rmva

or

cd ~/tests_atlas/pymva

root TMVAClassification.C

There is an example for batch job submittion

cd ~/tests_atlas/batch_jobs

qsub job.pbs

and map creating examples in R:

cd ~/tests_atlas/R/leaflet_example

Rscript leaflet_example.R

to see output map, or copy to your computer or

gthumb Rplot.png

or an example of visual prsentation

of measurement places on the World map

Rscript Stojic.R

New R example in ~/tests_atlas/R/Rtests

where you san check speed of sqlite DB access, (qsub job.pbs) or

see how to use multiple sqlite DBs by firstly unpacking some of sqlite db zips

in /home/dmaletic/AI_ATLAS_sqlite/ and use them by using script (Rscript razne_baze.R)

----

3. TMVA

In high-energy physics, with the search for ever smaller signals in ever larger data sets, it has become essential to extract a maximum of the available information from the data. Multivariate classification methods based on machine learning techniques have become a fundamental ingredient to most analyses. Also, the multivariate classifiers themselves have significantly evolved in recent years. Statisticians have found new ways to tune and combine classifiers to further gain in performance. Integrated into the analysis framework ROOT, TMVA is a toolkit that hosts a large variety of multivariate classification algorithms. Training, testing, performance evaluation, and the application of all available classifiers are carried out simultaneously via user-friendly interfaces. With version 4, TMVA has been extended to multivariate regression of a real-valued target vector. Regression is invoked through the same user interfaces as classification. TMVA 4 features more flexible data handling allowing one to arbitrarily form combined MVA methods. A generalized boosting method is the first realization benefiting from the new framework. [TMVA]

3.1 Data preprocessing

TMVA provides a preprocessing of the discriminating input variables or the training events prior to multivariate analysis. Preprocessing can be useful to reduce correlations among the variables, to transform their shapes into more appropriate forms, or to accelerate the response time of a method (event sorting).

TMVA implements five preprocessing transformations:

- variable normalisation;
- decorrelation via the square-root of the covariance matrix;
- decorrelation via a principal component decomposition;
- transformation of the variables into uniform distributions (“uniformization”); and
- transformation of the variables into Gaussian distributions (“Gaussianisation”).

3.1.1 Variable normalization

Minimum and maximum values for the variables to be transformed are determined from the training events and used to linearly scale input variables within [−1,1]. Such transformation allows direct comparisons between the MVA weights assigned to the variables, where large absolute weights may indicate strong separation power. Normalization may also render minimization processes, such as the adjustment of neural network weights, more effective.

3.1.2 Variable decorrelation

Linear correlations, measured in the training sample, can be taken into account in a straightforward manner through computing the square-root of the covariance matrix.

TMVA computes the square-root matrix by means of diagonalizing the (symmetric) covariance matrix

$$D = S^T C S \Rightarrow C' = S\sqrt{D}S^T$$

where D is a diagonal matrix, and where the matrix S is symmetric. The linear decorrelation of the selected variables is then obtained by multiplying the initial variable tuplex by the inverse of the square-root matrix:

$$x \to C'^{-1} x$$

The decorrelation is complete only for linearly correlated and Gaussian distributed variables. In the situations where these requirements are not fulfilled, only little additional information can be recovered by the decorrelation procedure. For highly nonlinear problems, the performance may even become worse when applying linear decorrelation. Nonlinear methods without a prior variable decorrelation should be used in such cases.

3.1.3 Principal component decomposition

Principal component decomposition is a linear transformation that rotates a sample of data points such that the maximum variability is visible. It thus identifies the most important gradients. In the principal component analysis (PCA)-transformed coordinate system, the largest variance by any projection of the data lies on the first coordinate (denoted the first principal component), the second largest variance on the second coordinate, and so on. PCA can thus be used to reduce the dimensionality of a problem (initially given by the number of input variables) by removing dimensions with insignificant variance. This corresponds to keeping lower-order principal components and ignoring higher-order ones.

The tuples $\boldsymbol{x}_U^{PC}(i) = (x_{U,1}^{PC}(i), \ldots, x_{U,n_{var}}^{PC}(i))$ of principal components of a tuple of input variables $x(i) = (x_1(i), \ldots, x_{nvar}(i))$ measured for the event, for signal (U=S) and background (U=B), are obtained by the transformation:

$$x_{U,k}^{PC}(i) = \sum_{l=1}^{n_{var}} (x_{U,l}(i) - \bar{x}_{U,l}) v_{U,l}^{(k)}, \; \forall k = 1, n_{var}$$

The tuples $\bar{\boldsymbol{x}}_U$ and $\boldsymbol{v}_U^{(k)}$ are the sample means and eigenvectors, respectively. The matrix of eigenvectors $V_U = (\boldsymbol{v}_U^{(1)}, \ldots, \boldsymbol{v}_U^{(n_{var})})$ obeys the relation $C_U \cdot V_U = D_U \cdot V_U$ where C is the covariance matrix of the sample U, and D_U is the tuple of eigenvalues.

3.1.4 Uniform and Gaussian transformation of variables

The decorrelation methods described above require linearly correlated and Gaussian distributed input variables. In real-life applications this is, however, rarely the case. One may hence transform the variables prior to their decorrelation such that their distributions become Gaussian. The corresponding transformation function is conveniently separated into two steps: first, transform a variable into a

uniform distribution; second, use the inverse error function to transform the uniform distribution into a Gaussian shape with zero mean and unity width.

3.2 MVA methods

The common goal of all TMVA discriminators is to determine an optimal separating function in the multivariate space of all input variables. The Fisher discriminant solves this analytically for the linear case, function discriminant analysis (FDA) provides an intermediate solution to solve relatively simple or partially nonlinear problems, while artificial neural networks, support vector machines, or boosted decision trees provide nonlinear approximations with, in principle, an arbitrary precision if enough training statistics is available and the chosen architecture is flexible enough.

3.2.1 Rectangular cut optimization

The simplest and most common classifier for selecting signal events from a mixed sample of signal and background events is the application of an ensemble of rectangular cuts on discriminating variables. Unlike all other classifiers in TMVA, the cut classifier returns only a binary response (signal or background). The optimization of cuts performed by TMVA maximizes the background rejection at given signal efficiency and scans over the full range of the latter quantity. For each variable, statistical properties, like mean, root-mean-squared (RMS), or variable ranges, are computed to guide the search for optimal cuts. Cut optimization requires an estimator that quantifies the goodness of a given cut ensemble. Maximizing this estimator minimizes (maximizes) the background efficiency, εB (background rejection, rB= 1− εB) for each signal efficiency εS. All optimization methods (fitters) act on the assumption that one minimum and one maximum requirement on each variable is sufficient to optimally discriminate signal from the background (i.e., the signal is clustered). If this is not the case, the variables must be transformed prior to the cut optimization to make them compliant with this assumption. For a given cut ensemble, the signal and background efficiencies are derived by counting the training events that pass the cuts and dividing the numbers found by the original sample sizes.

3.2.2 Projective likelihood estimator (PDE approach)

The method of maximum likelihood builds a model based on probability density functions (PDF) that reproduces the input variables for a signal and background. For a given event, the likelihood for being of signal type is obtained by multiplying the signal probability densities of all input variables, which are assumed to be independent, and normalizing this by the sum of the signal and background likelihoods. Because correlations among the variables are ignored, this PDE approach is also called “naive Bayes estimator”.

The likelihood ratio $y_L(i)$ for event *i* is defined by:

$$y_L(i) = \frac{L_S(i)}{L_S(i) + L_B(i)}$$

where $L_{S(B)}(i) = \prod_{k=1}^{n_{var}} p_{S(B),k}(x_k(i))$ and $p_{S(B),k}$ is the signal (background) PDF for the *k*th input variable x_k.

The PDFs are normalized as:

$$\int_{-\infty}^{+\infty} p_{S(B),k}(x_k)\,dxk = 1, \forall\ k.$$

Both the training and the application of the likelihood classifier are very fast operations that are suitable for large data sets. The performance of the classifier relies on the accuracy of the likelihood model. Because high fidelity PDF estimates are mandatory, sufficient training statistics is required to populate the tails of the distributions.

3.2.3 Multidimensional likelihood estimator (PDE range-search approach)

This method is a generalization of the projective likelihood classifier described previously to n_{var} dimensions, where n_{var} is the number of input variables. If the multidimensional PDF for signal and background (or regression data) is known, this classifier would exploit the full information contained in the input variables and would hence be optimal. In practice however, huge training samples are necessary to sufficiently populate the multidimensional phase space.

Kernel estimation methods may be used to approximate the shape of the PDF for finite training statistics. A simple probability density estimator denoted PDE range search, or PDE-RS, a variant of the k-nearest neighbour classifier, is used. The PDE for a given test event (discriminant) is obtained by counting the (normalized) number of training events that occur in the ”vicinity” of the test event. The classification of the test event may then be conducted based on the majority of the nearest training events. Then n_{var}-dimensional volume that encloses the ”vicinity” is user-defined and can be adaptive. A search method based on sorted binary trees is used to reduce the computing time for the range search. To enhance the sensitivity within the volume, kernel functions are used to weight the reference events according to their distance from the test event.

As opposed to many of the more sophisticated data-mining approaches, which tend to present the user with a ”black box”, PDE-RS is simple enough so the algorithm can be easily traced and tuned by hand. PDE-RS can yield competitive performance if the number of input variables is not too large and the statistics of the training sample is enough. In particular, it naturally deals with complex nonlinear variable correlations, the reproduction of which may, for example, require involved neural network architectures. PDE-RS is a slowly responding classifier. Only the training, i.e., the fabrication of the binary tree is fast, which is usually not the critical part. The necessity to store the entire binary tree in memory to avoid accessing virtual memory limits the number of training events that can effectively be used to model the multidimensional PDF.

3.2.4 Likelihood estimator using self-adapting phase-space binning (PDE-Foam)

The PDE-Foam method is an extension of PDE-RS, which divides the multi-dimensional phase space in a finite number of hyper-rectangles (cells) of constant event density. This “foam” of cells is filled with averaged probability density information sampled from the training data. For a given number of cells, the binning algorithm adjusts the size and position of the cells inside the multi-dimensional phase space based on a binary split algorithm that minimizes the variance of the event density in the cell. The binned event density information of the final foam is stored in cells, organized in a binary tree, to allow fast and memory-efficient storage and retrieval of the event density information necessary for classification or regression. The implementation of PDE-Foam is based on the Monte-Carlo integration package TFoam included in ROOT. In classification mode, PDE-Foam forms bins of the similar density of signal and background events or the ratio of signal to background. In the regression mode, the algorithm determines cells with small varying regression targets. In the following, the term density (ρ) is used for the event density in the case of classification or the target variable density in the case of regression.

Like PDE-RS, this method is a powerful classification tool for problems with highly non-linearly correlated observables. Furthermore, PDE-Foam is a fast-responding classifier, because of its limited number of cells, independent of the size of the training samples. An exception is a multi-target regression with Gauss kernel because the time scales with the number of cells squared. Also, the

training can be slow, depending on the number of training events and the number of cells one wishes to create.

3.2.5 k-Nearest Neighbour (k-NN) Classifier

Similar to PDE-RS, the k-nearest neighbour method compares an observed (test) event to reference events from a training data set. However, unlike PDE-RS, which in its original form uses a fixed-sized multidimensional volume surrounding the test event and in its augmented form resizes the volume as a function of the local data density, the k-NN algorithm is intrinsically adaptive. It searches for a fixed number of adjacent events, which then defines a volume for the metric used. The k-NN classifier has the best performance when the boundary that separates signal and background events has irregular features that cannot be easily approximated by parametric learning methods.

The k-NN algorithm searches for k events that are closest to the test event. Closeness is thereby measured using a metric function. The simplest metric choice is the Euclidean distance:

$$R = \left(\sum_{i=1}^{n_{var}} |x_i - y_i|^2\right)^{\frac{1}{2}}$$

where n_{var} is the number of input variables used for the classification, x_{i} are coordinates of an event from a training sample and y_i are variables of an observed test event. The k events with the smallest values of R are the k-nearest neighbours. The value of k determines the size of the neighbourhood for which a probability density function is evaluated. Large values of k do not capture the local behavior of the probability density function.

The simplest implementation of the k-NN algorithm would store all training events in an array. The classification would then be performed by looping over all stored events and finding the k-nearest neighbours.

The k-NN algorithm in TMVA also implements a simple multi-dimensional (multi-target) regression model. For a test event, the algorithm finds the k-nearest neighbours using the input variables, where each training event contains a regression value. The predicted regression value for the test event is the weighted average of the regression values of the k-nearest neighbours.

3.2.6 H-Matrix discriminant

The origins of the H-Matrix approach date back to 1936 and works of Fisher and Mahalanobis in the context of Gaussian classifiers. It discriminates one class (signal) of a feature vector from another (background). The correlated elements of the vector are assumed to be Gaussian distributed, and the inverse of the covariance matrix is the H-Matrix. A multivariate $\chi 2$ estimator is built to exploit differences in the mean values of the vector elements between the two classes for discrimination. The H-Matrix classifier, as it is implemented in TMVA, is equal or less performing than the Fisher discriminant and has been only included for completeness.

3.2.7 Fisher discriminants (linear discriminant analysis)

The method of Fisher discriminants selects the event in a transformed variable space with zero linear correlations, by distinguishing the mean values of the signal and background distributions. The linear discriminant analysis determines an axis in the (correlated) hyperspace of the input variables such that, when projecting the output classes (signal and background) up on this axis, they are pushed as far as possible away from each other, while events of a same class are confined in a close vicinity. The linearity property of this classifier is reflected in the metric with which ”far apart” and ”close vicinity” are determined: the covariance matrix of the discriminating variable space.

The classification of the events in signal and background classes relies on the following characteristics: the overall sample means $\bar{x}_k$ for each input variable $k = 1, ..., n_{var}$, the class-specific sample means $\bar{x}_{S(B),k}$, and the total covariance matrix C of the sample. The covariance matrix can be decomposed into the sum of a within-(W) and a between-class matrix(B). They respectively describe the dispersion of events relative to the means of their own class (within-class matrix) and relative to the overall sample means (between-class matrix).

In spite of the simplicity of the classifier, Fisher discriminants can be competitive with likelihood and non-linear discriminants in certain cases. In particular, Fisher discriminants are optimal for Gaussian distributed variables with linear correlations. On the other hand, no discrimination is achieved when a variable has the same sample mean for signal and background, even if the shapes of the distributions are very different. Thus, Fisher discriminants often benefit from suitable transformations of the input variables. For example, the simple transformation x→|x| can render this variable powerful for the use in a Fisher discriminant.

3.2.8 Linear discriminant analysis (LD)

The linear discriminant analysis provides data classification using a linear model, where linear refers to the discriminant function y(**x**) being linear in the parameters β, $y(\boldsymbol{x}) = \boldsymbol{x}^T\beta+\beta_0$, where β_0 (denoted the bias) is adjusted so that $y(\mathbf{x})\geq 0$ for signal and $y(\mathbf{x})<0$ for the background. It can be shown that this is equivalent to the Fisher discriminant, which seeks to maximize the ratio of between-class variance to within-class variance by projecting the data onto a linear subspace.

The LD is optimal for Gaussian distributed variables with linear correlations and can be competitive with likelihood and nonlinear discriminants in certain cases. No discrimination is achieved when a variable has the same sample mean for signal and background, but the LD can often benefit from suitable transformations of the input variables. For example, the simple transformation x→|x| renders this variable powerful for use with LD.

3.2.9 Function discriminant analysis (FDA)

The common goal of all TMVA discriminators is to determine an optimal separating function in the multivariate space of all input variables. The Fisher discriminant solves this analytically for the linear case, while artificial neural networks, support vector machines, or boosted decision trees provide nonlinear approximations with, in principle, an arbitrary precision if enough training statistics is available and the chosen architecture is flexible enough. The function discriminant analysis (FDA) provides an intermediate solution to the problem with the aim to solve relatively simple or partially nonlinear problems. The user provides the desired function with adjustable parameters via the configuration option string and FDA fits the parameters to it, requiring the function value to be as close as possible to the real value (to 1 for signal and 0 for background in classification). Its advantage over the more involved and automatic nonlinear discriminators is the simplicity and transparency of the discrimination expression. A shortcoming that FDA will underperform for involved problems with complicated, phase space-dependent nonlinear correlations.

The parsing of the discriminator function employs ROOT's TFormula class, which requires that the expression complies with its rules. For a simple formula with a single global fit solution, Minuit will be the most efficient fitter. However, if the problem is complicated, highly nonlinear, and/or has a non-unique solution space, more involved fitting algorithms may be required. In that case, the Genetic Algorithm combined or not with a Minuit converger should lead to the best results. After fit convergence, FDA prints the fit results (parameters and estimator value) as well as the discriminator expression used on standard output. The smaller the estimator value, the better the solution found. The normalized estimator is given by:

$$\varepsilon = \frac{1}{W_S}\sum_{i=1}^{N_S}(F(\boldsymbol{x}_i)-1)^2 w_i + \frac{1}{W_B}\sum_{i=1}^{N_B}F(\boldsymbol{x}_i)^2 w_i$$

for classification, and:

$$\varepsilon = \frac{1}{W}\sum_{i=1}^{N}(F(\boldsymbol{x}_i)-\boldsymbol{t}_i)^2 w_i$$

for regression

For classification, the first (second) sum is over the signal (background) training events, and for regression, it is over all training events, $F(\boldsymbol{x}_\mathrm{i})$ is the discriminator function, $\boldsymbol{x}_\mathrm{i}$ is the tuple of the n_var input variables for event i, w_i is the event weight, $\boldsymbol{t}_\mathrm{i}$ the tuple of training regression targets, $W_{\mathrm{S(B)}}$is the sum of all signal (background) weights in the training sample, and W the sum over all training weights.

The FDA performance depends on the complexity and fidelity of the user-defined discriminator function. As a general rule, it should be able to reproduce the discrimination power of any linear discriminant analysis. To reach into the nonlinear domain, it is useful to inspect the correlation profiles of the input variables and add quadratic and higher polynomial terms between variables as necessary. Comparison with more involved nonlinear classifiers can be used as a guide.

3.2.10 Artificial Neural Networks (nonlinear discriminant analysis)

An Artificial Neural Network (ANN) is, generally speaking, any simulated collection of inter-connected neurons, with each neuron producing a certain response at a given set of input signals. By applying an external signal to some (input) neurons, the network is put into a defined state that can be measured from the response of one or several (output) neurons. One can therefore view the neural network as a mapping from a space of input variables x_1, ..., x_{nvar} onto a one-dimensional (*e.g.* in case of a signal-versus-background discrimination problem) or multi-dimensional space of output variables y_1, ..., y_{mvar}. The mapping is nonlinear if at least one neuron has a nonlinear response to its input.

In TMVA four neural network implementations are available to the user. The first was adapted from a FORTRAN code developed at the Universite Blaise Pascal in Clermont-Ferrand. The second is the ANN implementation that comes with ROOT. The third is a newly developed neural network (denoted MLP) that is faster and more flexible than the other two and is the recommended neural network to use with TMVA. The fourth implementation is a highly optimized one which provides functionality to perform the training on multi-core and GPU architectures. This implementation has been developed specifically for the training of very complex neural networks with several hidden layers, so called deep neural networks. All four neural networks are feed-forward multilayer perceptrons.

The behaviour of an artificial neural network is determined by the layout of the neurons, the weights of the inter-neuron connections, and by the response of the neurons to the input, described by the *neuron response function* ρ.

Multilayer Perceptron

While in principle a neural network with n neurons can have n^2 directional connections, the complexity can be reduced by organizing the neurons in layers and only allowing direct connections from a given layer to the following layer, Figure 2.

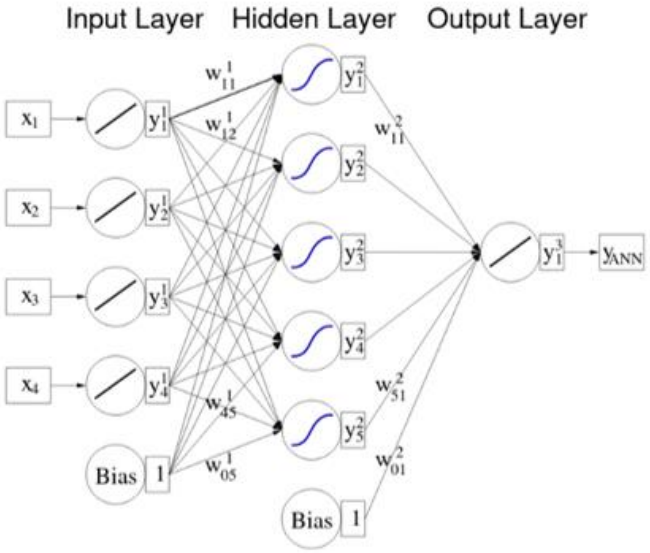


Figure 2: Multilayer perceptron with one hidden layer.

This kind of neural network is termed multilayer perceptron – all neural network implementations in TMVA are of this type. The first layer of a multilayer perceptron is the input layer, the last one the output layer, and all others are hidden layers. For a classification problem with n_{var} input variables the input layer consists of n_{var} neurons that hold the input values, x_1, ..., x_{nvar}, and one neuron in the output layer that holds the output variable, the neural net estimator y_{ANN}.

The neuron response function ρ maps the neuron input i_1, ..., i_n onto the neuron output. Often it can be separated into a $R^n \to R$ *synapse function* k, and a R→R neuron *activation function* α, so that $\rho = \alpha \circ k$. The functions k and α can have the following forms:

$$k:(y_1^{(l)},..,y_n^{(l)}|w_{0j}^{(l)},..,w_{nj}^{(l)}) \to \begin{cases} w_{0j}^{(l)} + \sum_{i=1}^{n} y_i^{(l)} w_{ij}^{(l)} & \text{Sum,} \\ w_{0j}^{(l)} + \sum_{i=1}^{n} (y_i^{(l)} w_{ij}^{(l)})^2 & \text{Sum of squares,} \\ w_{0j}^{(l)} + \sum_{i=1}^{n} |y_i^{(l)} w_{ij}^{(l)}| & \text{Sum of absolutes,} \end{cases}$$

$$\alpha:x \to \begin{cases} x & \text{Linear,} \\ \frac{1}{1+e^{-kx}} & \text{Sigmoid,} \\ \frac{e^x - e^{-x}}{e^x + e^{-x}} & \text{Tanh,} \\ e^{-x^2/2} & \text{Radial} \end{cases}$$

When building a network two rules should be kept in mind. The first is the theorem by Weierstrass, which if applied to neural nets, ascertains that for a multilayer perceptron a single hidden layer is sufficient to approximate a given continuous correlation function to any precision, provided that a sufficiently large number of neurons is used in the hidden layer. If the available computing power and the size of the training data sample suffice, one can increase the number of neurons in the hidden layer until the optimal performance is reached. It is likely that the same performance can be achieved with a network of more than one hidden layer and a potentially much smaller total number of hidden neurons. This would lead to a shorter training time and a more robust network.

In the tests, the MLP and ROOT networks performed equally well, however with a clear speed advantage for the MLP. The Clermont-Ferrand neural net exhibited worse classification performance in these tests, which is partly due to the slow convergence of its training (at least 10k training cycles are required to achieve approximately competitive results).

Deep Learning

The Deep Learning module in TMVA provides the support to build several deep learning architectures such as Deep Neural Networks (DNN), Convolutional networks (CNN), and Recurrent networks (RNN). The purpose of this module is to provide optimized implementations of these networks that can be efficiently trained on modern multi-core and GPU hardware architectures. For this, the implementation provides two backends for training and evaluating deep learning models. The CPU backend uses multithreading to perform the training in parallel on multi-core CPU architectures and it can make use of efficient multithreaded BLAS implementations such as OpenBLAS and the Intel TBB library. The GPU backend can be used to perform the training on CUDA-capable GPU architectures. In the case of Convolutional and Recurrent network models the implementation uses the low level CuDNNlibrary of CUDA for optimal performances.

A **deep neural network** (DNN) is an artificial neural network with several hidden layers and a large number of neurons in each layer. Recent developments in machine learning have shown that these networks are capable of learning complex, non-linear relations when trained on a sufficiently large amount of training data. The deep neural network implementation provides an optimized implementation of feed-forward multilayer perceptrons that can be efficiently trained on modern multi-core and GPU architectures. Note that previous ROOT/TMVA versions (version<6.22) provide for Deep Neural Networks a method TMVA::kDNN, implementing only fully connected layers. Since ROOT version 6.18 it is recommended to use the new TMVA::kDL method, which supports not only dense layer as in the previous implementation, but also new type of layers as convolutional and recurrent, allowing to build convolutional and recurrent networks.

Convolutional Neural Networks are a popular variation of the feed-forward networks. By making the explicit assumption that the input is an image, they manage to outperform conventional networks by significantly reducing the number of learnable parameters through a feature known as parameter sharing. Instead of connecting every neuron of the current layer with every neuron of the previous layer as in conventional dense layers, we only learn the parameters for a set of a small kernels (typically 3x3 or 5x5) which slide over the input. The size of each kernel, as well as the number of pixels for each vertical or horizontal shift, are user defined meta-parameters. An important detail is that while the model assumes spatial invariance with respect to the height and width, no such assumption is made with respect to the image depth. This asymmetry implies that at each receptive field, the kernel is connected with every slice of the input volume. In order to reduce overfitting, we sometimes include a pooling layer between subsequent convolutional layers. This layer simply downsamples the input, typically by outputting the maximum value within its receptive field. This does not only reduce the number of neurons, but also helps reduce overfitting.

Recurrent neural network is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Different types of recurrent networks exist. TMVA implementation provides the support for 3 types of recurrent layers: Simple Recurrent layer (Vanilla RNN), LSTM (Long Short-Term Memory) layer and GRU (Gate Recurrent unit).

3.2.11 Support Vector Machine (SVM)

In the early 1960s a linear support vector method has been developed for the construction of separating hyperplanes for pattern recognition problems. It took 30 years before the method was generalized to nonlinear separating functions and for estimating real-valued functions (regression). At that moment it became a general-purpose algorithm, performing classification and regression tasks which can compete with neural networks and probability density estimators. Typical applications of SVMs include text categorization, character recognition, bio-informatics, and face detection. The main idea of the SVM approach to classification problems is to build a hyperplane that separates signal and background vectors (events) using only a minimal subset of all training vectors (support vectors). The position of the hyperplane is obtained by maximizing the margin (distance) between it and the support vectors. The extension to nonlinear SVMs is performed by mapping the input vectors onto a higher dimensional feature space in which signal and background events can be separated by a linear procedure using an optimally separating hyperplane.

3.2.12 Boosted Decision and Regression Trees

A decision (regression) tree is a binary tree structured classifier (regressor). Repeated left/right (yes/no) decisions are taken on one single variable at a time until a stop criterion is fulfilled. The phase space is split this way into many regions that are eventually classified as signal or background, depending on the majority of training events that end up in the final leaf node. In case of regression trees, each output node represents a specific value of the target variable. The boosting of a decision (regression) tree extends this concept from one tree to several trees which form a forest. The trees are derived from the same training ensemble by reweighting events, and are finally combined into a single classifier (regressor) which is given by a (weighted) average of the individual decision (regression) trees. Boosting stabilizes the response of the decision trees with respect to fluctuations in the training sample and is able to considerably enhance the performance with respect to a single tree.

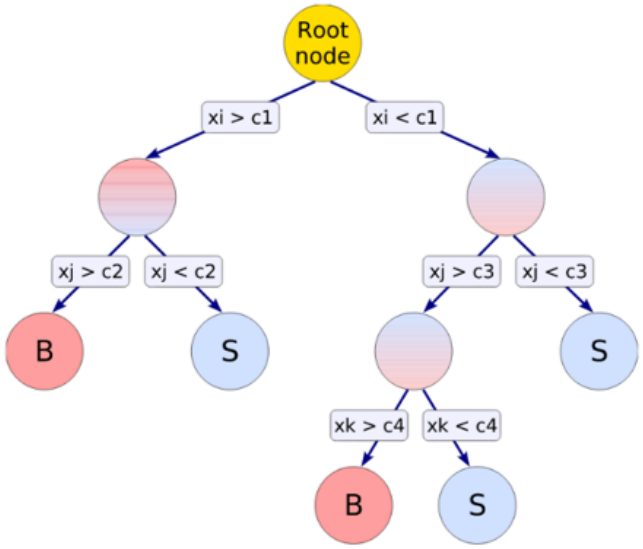


Figure 3: Schematic view of a decision tree. Starting from the root node, a sequence of binary splits using the discriminating variables x_i is applied to the data. Each split uses the variable that at this node gives the best separation between signal and background when being cut on. The same variable may thus be used at several nodes, while others might not be used at all. The leaf nodes at the bottom end of the tree are labeled "S" for signal and "B" for background depending on the majority of events that end up in the respective nodes.

Decision trees are well known classifiers that allow a straightforward interpretation as they can be visualized by a simple two-dimensional tree structure. They are in this respect similar to rectangular cuts. However, whereas a cut-based analysis is able to select only one hypercube as region of phase

space, the decision tree is able to split the phase space into a large number of hypercubes, each of which is identified as either “signal-like” or “background-like”, or attributed a constant event (target) value in case of a regression tree. For classification trees, the path down the tree to each leaf node represents an individual cut sequence that selects signal or background depending on the type of the leaf node.

A shortcoming of decision trees is their instability with respect to statistical fluctuations in the training sample from which the tree structure is derived. For example, if two input variables exhibit similar separation power, a fluctuation in the training sample may cause the tree growing algorithm to decide to split on one variable, while the other variable could have been selected without that fluctuation. In such a case the whole tree structure is altered below this node, possibly resulting also in a substantially different classifier response. This problem is overcome by constructing a forest of decision trees and classifying an event on a majority vote of the classifications done by each tree in the forest. All trees in the forest are derived from the same training sample, with the events being subsequently subjected to so-called boosting, a procedure which modifies their weights in the sample. Boosting increases the statistical stability of the classifier and is able to drastically improve the separation performance compared to a single decision tree. However, the advantage of the straightforward interpretation of the decision tree is lost. While one can of course still look at a limited number of trees trying to interpret the training result, one will hardly be able to do so for hundreds of trees in a forest. Nevertheless, the general structure of the selection can already be understood by looking at a limited number of individual trees. In many cases, the boosting performs best if applied to trees (classifiers) that, taken individually, have not much classification power. These so called “weak classifiers” are small trees, limited in growth to a typical tree depth of as small as two, depending on the how much interaction there is between the different input variables. By limiting the tree depth during the tree building process (training), the tendency of overtraining for simple decision trees which are typically grown to a large depth and then pruned (process of cutting back a tree from the bottom up after it has been built to its maximum size), is almost completely eliminated.

3.2.13 Predictive learning via rule ensembles (RuleFit)

This classifier is a TMVA implementation of Friedman-Popescu’s RuleFit method. Its idea is to use an ensemble of so-called rules to create a scoring function with good classification power. Each rule r_i is defined by a sequence of cuts, such as:

$$r_1(x) = I(x_2<100.0)\cdot I(x_3>35.0),$$

$$r_2(x) = I(0.45< x_4<1.00)\cdot I(x_1>150.0),$$

$$r_3(x) = I(x_3<11.00)$$

where the x_i are discriminating input variables, and I(...) returns the truth of its argument. A rule applied on a given event is non-zero only if all of its cuts are satisfied, in which case the rule returns 1. The easiest way to create an ensemble of rules is to extract it from a forest of decision trees. Every node in a tree (except the root node) corresponds to a sequence of cuts required to reach the node from the root node and can be regarded as a rule. Linear combinations of the rules in the ensemble are created with coefficients (rule weights) calculated using a regularized minimization procedure. The resulting linear combination of all rules defines a score function which provides the RuleFit responsey RF(x). In some cases, a very large rule ensemble is required to obtain a competitive discrimination between signal and background. A particularly difficult situation is when the true (but unknown) scoring function is described by a linear combination of the input variables. In such cases, *e.g.*, a Fisher discriminant would perform well. To ease the rule optimization task, a linear combination of the input variables is added to the model. The minimization procedure will then select the appropriate coefficients for the rules and the linear terms.

3.3 PyMVA and RMVA interfaces

PyMVA and RMVA are interfaces for third-party MVA tools based on Python and R. They are created to make powerful external libraries easily accessible with a direct integration into the TMVA workflow.

All PyMVA and RMVA methods provide the same plug-and-play mechanisms as for other TMVA methods. For example, to use Python methods in PyMVA, the Keras method is booked, mainly similar to other TMVA methods. Keras (www.keras.io) is a high-level wrapper for the machine learning frameworks Theano (www.deeplearning.net/software/theano/) and TensorFlow (www.tensorflow.org), which are mainly used to set up deep neural networks.

In R, the most interesting for the ATLAS experiments is Gradient Boosting technique for performing supervised machine learning tasks, like classification and regression. The implementations of this technique can have different names, most commonly you encounter Gradient Boosting machines (abbreviated GBM) and XGBoost. XGBoost is particularly popular because it has been the winning algorithm in a number of recent Kaggle competitions.

Gradient Boosting is an ensemble learner which means that it will create a final model based on a collection of individual models. The predictive power of these individual models is weak and prone to overfitting, but combining many such weak models in an ensemble will lead to an overall much improved result. In Gradient Boosting machines, the most common type of weak model used is decision trees.

Another R implemented MVA method is C5.0. While there are numerous implementations of decision trees, one of the most well-known is the C5.0 algorithm. The C5.0 algorithm has become the industry standard for producing decision trees, because it does well for most types of problems directly out of the box. Compared to more advanced and sophisticated machine learning models (*e.g.,* Neural Networks and Support Vector Machines), the decision trees under the C5.0 algorithm generally perform nearly as well, but are much easier to understand and deploy.

4. Conclusions

/

References

[1] TMVA,
https://jnportal.ujn.gov.rs/

[2] Smartmontools official web page,
https://www.smartmontools.org/

[3] HPE Integrated Lights Out,
https://www.hpe.com/us/en/servers/integrated-lights-out-ilo.html

[4] Torque resource manager,
https://adaptivecomputing.com/cherry-services/torque-resource-manager/

[5] Maui scheduler, administrator’s guide,
http://docs.adaptivecomputing.com/maui/

[6] L. Hochstein, R. Moser, "Ansible: Up and Running, 2nd Edition", O'Reilly Media, August 2017.

[7] Lmod: A new environment module system,
https://lmod.readthedocs.io/en/latest/

[8] EasyBuild documentation,
https://docs.easybuild.io/en/latest/

[9] Github cuda_memtest repository,
https://github.com/ComputationalRadiationPhysics/cuda_memtest

[10] Memtest official web page,
https://www.memtest.org/

[11] HPL, a portable implementation of the High-Performance Linpack benchmark,
https://www.netlib.org/benchmark/hpl/

[12] Mellanox perftest package,
https://community.mellanox.com/s/article/perftest-package

[13] MeteoInfo framework for GIS application and scientific computation environment,
http://meteothink.org/