*arXiv physics/0703039*
*CERN-OPEN-2007-007*
*Document version 1.0*
*TMVA version 3.6*
*March 6, 2007*
*http:// tmva.sf.net*

# TMVA

**Toolkit for Multivariate Data Analysis with ROOT**

# Users Guide

A. Höcker, J. Stelzer, F. Tegenfeldt, H. Voss, K. Voss

*With contributions from*

A. Christov, S. Henrot-Versillé, M. Jachowski, A. Krasznahorkay Jr.,
Y. Mahalalel, X. Prudent, P. Speckmayer

**Abstract**

Multivariate machine learning techniques for the classification of data from high-energy physics (HEP) experiments have become standard tools in most HEP analyses. The multivariate classifiers themselves have significantly evolved in recent years, also driven by developments in other areas inside and outside science. TMVA is a toolkit integrated in ROOT which hosts a large variety of multivariate classification algorithms. They range from rectangular cut optimisation (using a genetic algorithm) and likelihood estimators, over linear and non-linear discriminants (neural networks), to sophisticated recent developments like boosted decision trees and rule ensemble fitting. TMVA organises the simultaneous training, testing, and performance evaluation of all these classifiers with a user-friendly interface, and expedites the application of the trained classifiers to the analysis of data sets with unknown sample composition.

**TMVA 3.6 – Toolkit for Multivariate Data Analysis with ROOT**



**Authors:**

Andreas Höcker (CERN, Switzerland), ⟨andreas.hoecker@cern.ch⟩,
Jörg Stelzer (CERN, Switzerland) ⟨joerg.stelzer@cern.ch⟩,
Fredrik Tegenfeldt (Iowa University, USA) ⟨fredrik.tegenfeldt@cern.ch⟩,
Helge Voss (MPI für Kernphysik Heidelberg, Germany) ⟨helge.voss@cern.ch⟩,
Kai Voss (University of Victoria, Canada) ⟨kai.voss@cern.ch⟩,
Asen Christov (Universität Freiburg, Germany) ⟨christov@physik.uni-freiburg.de⟩,
Sophie Henrot-Versillé (LAL Orsay, Francs) ⟨versille@lal.in2p3.fr⟩,
Matthew Jachowski (Stanford University, USA) ⟨jachowski@stanford.edu⟩,
Yair Mahalalel (Tel Aviv University, Israel) ⟨yair@mahalalel.com⟩,
Xavier Prudent (LAPP Annecy, Franec) ⟨prudent@lapp.in2p3.fr⟩,
Peter Speckmayer (CERN, Switzerland) ⟨speckmay@mail.cern.ch⟩,
and valuable contributions from other developers, please see acknowledgments.

# Contents

# 1 Introduction

The Toolkit for Multivariate Analysis (TMVA) provides a ROOT-integrated environment for the processing and parallel evaluation of sophisticated multivariate classification techniques.[1] TMVA is specifically designed to the needs of high-energy physics (HEP) applications,[2] but should not be restricted to these. The package includes:

- Rectangular cut optimisation (binary splits, Sec. 5.1)
- Projective likelihood estimator (Sec. 5.2)
- Multi-dimensional likelihood estimator (PDE range-search approach, Sec. 5.3)
- Linear discriminant analysis (H-Matrix and Fisher discriminants, Secs. 5.4, 5.5)
- Artificial neural networks (three different implementations, Sec. 5.6)
- Boosted/bagged decision trees (Sec. 5.7)
- Predictive learning via rule ensembles (RuleFit, Sec. 5.8)

The software package consists of object-oriented implementations in C++/ROOT for each of these discrimination techniques and provides training, testing and performance evaluation algorithms and visualization scripts. Detailed descriptions of all the TMVA classifiers and their options are given in Sec. 5. Their training and testing is performed with the use of user-supplied data sets in form of ROOT trees or text files, where each event can have an individual weight. The sample composition (event classification) in these data sets must be known. Preselection requirements and transformations can be applied on this data. TMVA supports the use of variable combinations and formulas, just as they are available for the `Draw` command of a ROOT tree.

TMVA works in transparent factory mode to guarantee an unbiased performance comparison between the classifiers: all classifiers see the same training and test data, and are evaluated following the same prescriptions within the same execution job. A *Factory* class organises the interaction between the user and the TMVA analysis steps. It performs preanalysis and preprocessing of the training data to assess basic properties of the discriminating variables used as input to the classifiers. The linear correlation coefficients of the input variables are calculated and displayed, and a preliminary ranking is derived (which is later superseded by classifier-specific variable rankings). The variables can be linearly transformed (individually for each classifier) into a non-correlated variable

[1] A classification problem corresponds in more general terms to a *discretised regression* problem. A regression is the process that estimates the parameter values of a function, which predicts the value of a response variable in terms of the values of other variables (the *input* variables).

[2] TMVA discriminates signal from background in data sets with unknown composition of these two samples. In frequent use cases the background (sometimes also the signal) consists of a variety of different populations with characteristic properties, which could call for classifiers with more than two discrimination classes. However, in practice it is usually possible to serialize background fighting by training individual classifiers for each background source, and applying consecutive requirements to these.

space or projected upon their principle components. To compare the signal-efficiency and background-rejection performance of the classifiers, the analysis job prints tabulated results for some benchmark values (see Sec. 3.1.7), besides other criteria such as a measure of the separation and the maximum signal significance. Smooth efficiency versus background rejection curves are stored in a ROOT output file, together with other graphical evaluation information. These results can be displayed using ROOT macros, which are conveniently executed via a graphical user interface that comes with the TMVA distribution (see Sec. 3.2). The TMVA training job runs alternatively as a ROOT script, as a standalone executable, where libTMVA.so is linked as a shared library, or as a python script via the PyROOT interface. Each classifier trained in one of these applications writes its configuration and training results in result ("weight") files, which consists of one text and one ROOT file.

A light-weight *Reader* class is provided, which reads and interprets the weight files (interfaced by the corresponding classifiers), and which can be included in any C++ executable, ROOT macro or python analysis job.

We have put emphasis on the clarity and functionality of the Factory and Reader interfaces to the user applications. All classifiers run with reasonable default configurations, so that for standard applications that do not require particular tuning, the user script for a full TMVA analysis will hardly exceed a few lines of code. For individual optimisation the user can (and should) customize the classifiers via configuration strings.

This manual introduces the TMVA training Factory and Reader interfaces, and describes design and implementation of the various multivariate classifiers. It is not the goal here to provide a general introduction to multivariate analysis techniques. Other excellent reviews exist on this subject (see, e.g., Refs. [1, 2, 3]). The document begins with a quick TMVA start reference in Sec. 2, and provides a more complete introduction to the TMVA design and its functionality in Sec. 3. Common tools used by several classifiers such as the transformation of input variables are discussed in Sec. 4. All the TMVA classifiers including their configurations and tuning options are described in Secs. 5.1-5.8.

### Copyrights and credits



Several similar combined multivariate classification ("data mining") efforts exist with rising importance in most fields of science and industry. In the HEP community the package *StatPatternRecognition* [4] is in use. Other codes exist, which however often treat only a subset of the most common MVAs [5]. The idea of parallel training and evaluation of MVA-based classification in HEP has been pioneered by the *Cornelius* package, developed by the Tagging Group of the BABAR Collaboration [6].

[3] For the BSD license, see http://tmva.sf.net/LICENSE.

# 2 TMVA Quick Start

To run TMVA it is not necessary to know much about its concepts or to understand the detailed functionality of the multivariate classifiers. Better, just begin with the quick start tutorial given below. One should note that the TMVA version obtained from the open source software platform Sourceforge.net (where TMVA is developed), and the one which is part of ROOT have a different directory structure for the example macros used for the tutorial. Wherever differences in command lines occur, they are given for both versions.

## 2.1 How to download and build TMVA

TMVA is maintained at Sourceforge.net (http://tmva.sf.net). The TMVA project is built upon ROOT (http://root.cern.ch/), so that for TMVA to run ROOT must be installed. Since ROOT version 5.11/06, TMVA comes as integral part of ROOT and can be used from the ROOT prompt without further preparation. For older ROOT versions or if the latest TMVA features are desired, the TMVA source code can be downloaded from Sourceforge.net. Since we do not provide prebuilt libraries for any platform, the library must be built by the user (see below). The source code can be either downloaded as a gzipped tar file or via CVS anonymous access

```
~> cvs -z3 -d:pserver:anonymous@tmva.cvs.sourceforge.net:/cvsroot/tmva \
   co -r V03-04-00 -P TMVA
```

Code Example 1: TMVA source code download via CVS. The latest code (CVS HEAD) can be downloaded by typing the same command without specifying a version: `cvs -d... -z3 co -P TMVA`.

While the source code is known to compile with VisualC++ on Windows (which is a requirement for ROOT), we do not provide project support for this platform yet. For Unix and most Linux flavours custom Makefiles are provided with the TMVA distribution, so that the library can be built by typing

```
~> cd TMVA
~/TMVA> source setup.sh # for c-shell family: source setup.csh
~/TMVA> cd src
~/TMVA/src> make
```

Code Example 2: Building the TMVA library under Linux/Unix using the provided Makefile. The `setup.[c]sh` script must be executed to ensure the correct setting of symbolic links and library paths required by TMVA.

After compilation, the library `~/TMVA/lib/libTMVA.so` should be present.

## 2.2 Version compatibility

TMVA can be run with any ROOT version above v4.02. The few occurring conflicts due to ROOT source code evolution after v4.02 are intercepted in TMVA via C++ preprocessor conditions.

## 2.3 The TMVA namespace

All TMVA classes are embedded in the namespace `TMVA`. For interactive access, or use in macros the classes must thus be preceded by `TMVA::`, or one may use the command `using namespace TMVA` instead.

## 2.4 Example jobs

TMVA comes with example jobs for the training phase (this phase actually includes training, testing and evaluation) using the TMVA *Factory*, as well as the application of the training results in a classification analysis using the TMVA *Reader*. The first task is performed in the program `TMVAnalysis`, and the second in `TMVApplication`.

In the ROOT version of TMVA, the macros `TMVAnalysis.C` and `TMVApplication.C` are located in the directory `$ROOTSYS/tmva/test/`.

In the Sourceforge.net version, the macros `TMVAnalysis.C` and `TMVApplication.C` are located in `~/TMVA/macros`. At Sourceforge.net we also provide these examples in form of the C++ executables `TMVAnalysis.cxx` and `TMVApplication.cxx`, which are located in `~/TMVA/examples`. To create the executables, type `cd ~/TMVA/example; make`, and then simply execute them by typing `./TMVAnalysis` and `./TMVApplication`. To illustrate how TMVA can be used in a python script via PyROOT we also provide the script `TMVAnalysis.py` located in `~/TMVA/python`, which again has the same functionality as the macro `TMVAnalysis.C`.

## 2.5 Running the example

The easiest way to get started with TMVA is to run the `TMVAnalysis.C` example macro. It uses an academic toy data set for training and testing, which consists of four linearly correlated, Gaussian distributed discriminating input variables, with different sample means for signal and background. All classifiers are trained, tested and evaluated using the toy dataset in the same way the user is expected to proceed for his or her own data. It is a valuable exercise to look at the example file in more detail. Most of the command lines therein should be self explaining, and one will easily find how they need to be customized to run TMVA on a real use case. A detailed description is given in Sec. 3.

The toy data set used by the example is included in the Sourceforge.net download. For the ROOT distribution, the example macro `TMVAnalysis.C` automatically fetches the data file from the web using the corresponding `TFile` constructor `TFile::Open("http://root.cern.ch/files/tmva_example.root")`. The example ROOT macro can be run from any

designated test directory ~/workdir, after adding the macro directory to ROOT's macro search path:

```
~/workdir> echo "Unix.*.Root.MacroPath: ~/TMVA/macros" >> .rootrc
~/workdir> root -l ~/TMVA/macros/TMVAnalysis.C
```

Code Example 3: Running the example `TMVAnalysis.C` using the Sourceforge.net version of TMVA.

```
~/workdir> echo "Unix.*.Root.MacroPath: $ROOTSYS/tmva/test" >> .rootrc
~/workdir> root -l $ROOTSYS/tmva/test/TMVAnalysis.C
```

Code Example 4: Running the example `TMVAnalysis.C` using the ROOT version of TMVA.

The training job provides formatted output logging containing analysis information such as: signal and background linear correlation matrices for the input variables, variable ranking, summaries of the classifier configurations, goodness-of-fit evaluation for PDFs (if requested), signal and background correlations between the various classifiers, their signal/background-likeness decision overlap, signal efficiencies at benchmark background rejection rates as well as other performance estimators, and overtraining validation output.

### 2.5.1 Displaying the results

Besides so-called "weight" files containing the classifier-specific training results, TMVA also provides a variety of control and performance plots that can be displayed via a set of ROOT macros available in ~/TMVA/macros/ or $ROOTSYS/tmva/test/ for the Sourceforge.net and ROOT distributions of TMVA, respectively. The macros are summarized in Table 1. At the end of the example job a graphical user interface (GUI) is displayed, which conveniently allows to run these macros (see Fig. 1).

Examples for plots produced by these macros are given in Figs. 3-5. The distributions of the input variables for signal and background according to our example job are shown in Fig. 2. It is useful to quantify the correlations between the input variables. These are drawn in form of a scatter plot with the superimposed profile for two of the input variables in Fig. 3 (upper left). As will be discussed in Sec. 4, TMVA allows to perform a linear decorrelation transformation of the input variables prior to the classifier training. The result of such decorrelation is shown at the upper right hand plot of Fig. 3. The lower plots display the linear correlation coefficients between all input variables, for the signal and background training samples.

Figure 4 shows some of the classifier output distributions for signal and background events from the test sample. By TMVA convention, signal (background) events accumulate at large (small) classifier output values. Hence, cutting on the output and retaining the events with $y_{\mathrm{MVA}}$ larger than the cut requirement selects signal samples with efficiencies

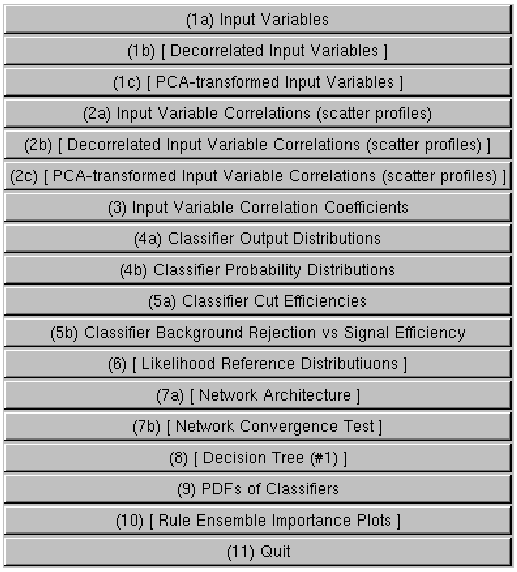


Figure 1: Graphical user interface (GUI) to execute macros displaying training, test and evaluation results (see Table 1). The GUI can be launched manually by executing the scripts `~/TMVA/macros/TMVAGui.C` (Sourceforge.net version) or `$ROOTSYS/tmva/test/TMVAGui.C` (ROOT version). In short, the buttons behave as follows: (1a) plots the signal and background distributions of the input variables (training sample), (1b) the same after decorrelation transformation, (1c) the same after PCA decorrelation (these latter two buttons only display results if the corresponding transformations have been requested by at least one classifier), (2a–c) scatters (with superimposed profiles) of all pairs of input variables for signal and background for the three transformation types (training sample), (3) linear correlation coefficients between the input variables for signal and background (training sample), (4a) signal and background distributions for the trained classifiers (test sample), (4b) the corresponding probability distributions, (5a) signal and background efficiencies (and purities assuming an equal number of signal and background events) as a function of the cut on the classifier outputs, (5b) background rejection versus signal efficiency obtained when cutting on the classifier outputs (ROC curve, from the test sample). The following buttons launch classifier-specific macros: (6) reference distributions (PDFs) used for the likelihood classifier compared to the training data, (7a) architecture of the MLP neural network, (7b) convergence of the MLP error parameter for the training and test samples (check for overtraining), (8) plots a sketch of the first decision tree in the forest, (9) compares the classifier PDFs to the training data, (10) plots the importance for the RuleFit classifier, and (11) quits the GUI. Titles in brackets indicate actions that can only be taken if the corresponding transformations or classifiers have been applied/used during the training.

and purities that respectively decrease and increase with the cut value. The resulting relations between background rejection versus signal efficiency are shown in Fig. 5 for all classifiers that were used in the example macro. This plot belongs to the class of *Receiver Operating Characteristic* diagrams, which in its standard form shows the true positive rate versus the false positive rate for the different possible cutpoints of a hypothesis test.

More analysis plots are available, in particular those that validate specific classifiers. For example, the macro `likelihoodrefs.C` compares the probability density functions used by the likelihood classifier to the normalised variable distributions of the training sample. It is also possible to visualize the MLP neural network architecture and to draw decision trees (see Table 1 for a complete summary of the available scripts).

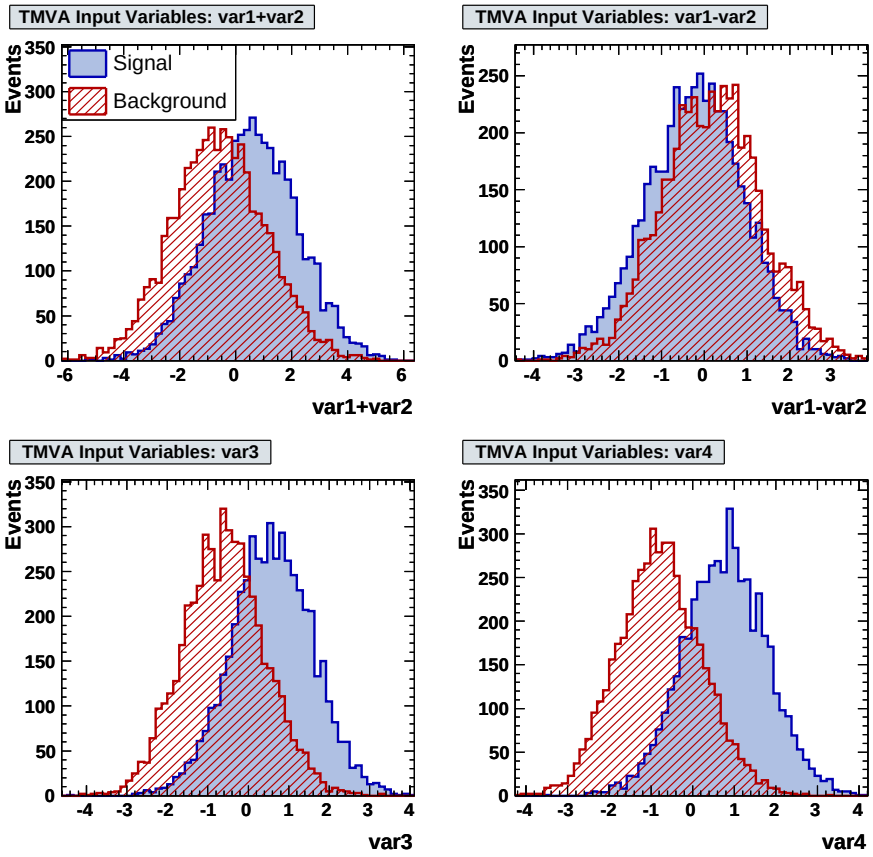


Figure 2: Example plots for input variable distributions.

# 3 Using TMVA

A typical TMVA analysis consists of two independent phases: the *training* phase, where the multivariate classifiers are trained, tested and evaluated, and an *application* phase, where selected classifiers are applied to the concrete classification problem they have been trained for. An overview of the code flow for these two phases as implemented in the examples `TMVAnalysis.C` and `TMVApplication.C` (see Sec. 2.4), is sketched in Fig. 6.

In the training phase, the communication of the user with the data sets and the classifiers is performed via a `Factory` object, created at the beginning of the program. The TMVA Factory provides member functions to specify the training and testing data sets, to register the discriminating input variables, and to book the multivariate classifiers. After the configuration the Factory calls for training, testing and the evaluation of the booked classifiers. Classifier-specific result ("weight") files are created after the training phase.

The application of training results to a data set with unknown sample composition is

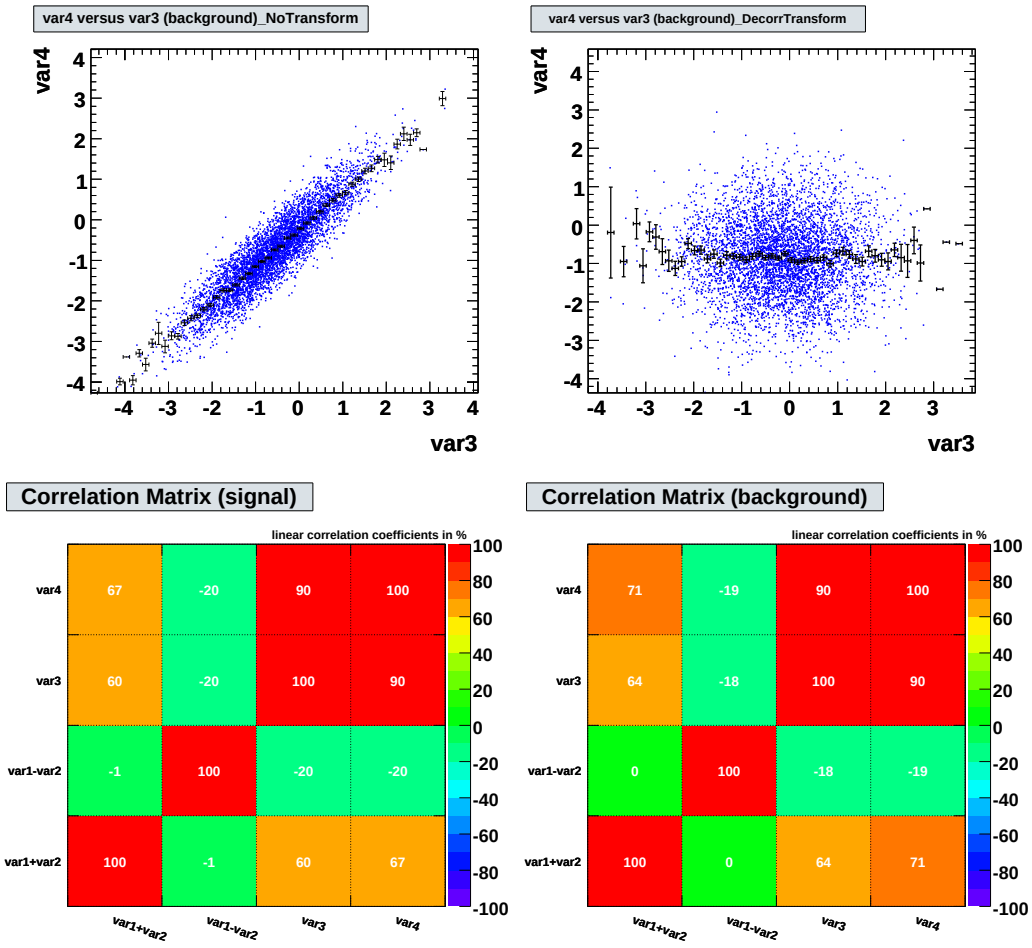


Figure 3: Correlation between input variables. Upper left: correlations between var3 and var4 for the background training sample. Upper right: the same after applying a decorrelation transformation (see Sec. 4.1.1). Lower plots: linear correlation coefficients for the signal and background training samples.

governed by the `Reader` object. During initialization, the user registers the input variables[4] together with their local memory addresses, and books the classifiers that were found to be the most appropriate ones during the training phase. As booking argument, the bulk name of the weight file is given, which provides for each of the classifiers full and consistent configuration according to the training results. Within the event loop, the input variables are updated for each event, and the selected classifier outputs are computed.

## 3.1 The Factory

The TMVA training phase begins by instantiating a `Factory` object

[4]This somewhat redundant operation is required to verify the correspondence between the Reader analysis and the weight files used.

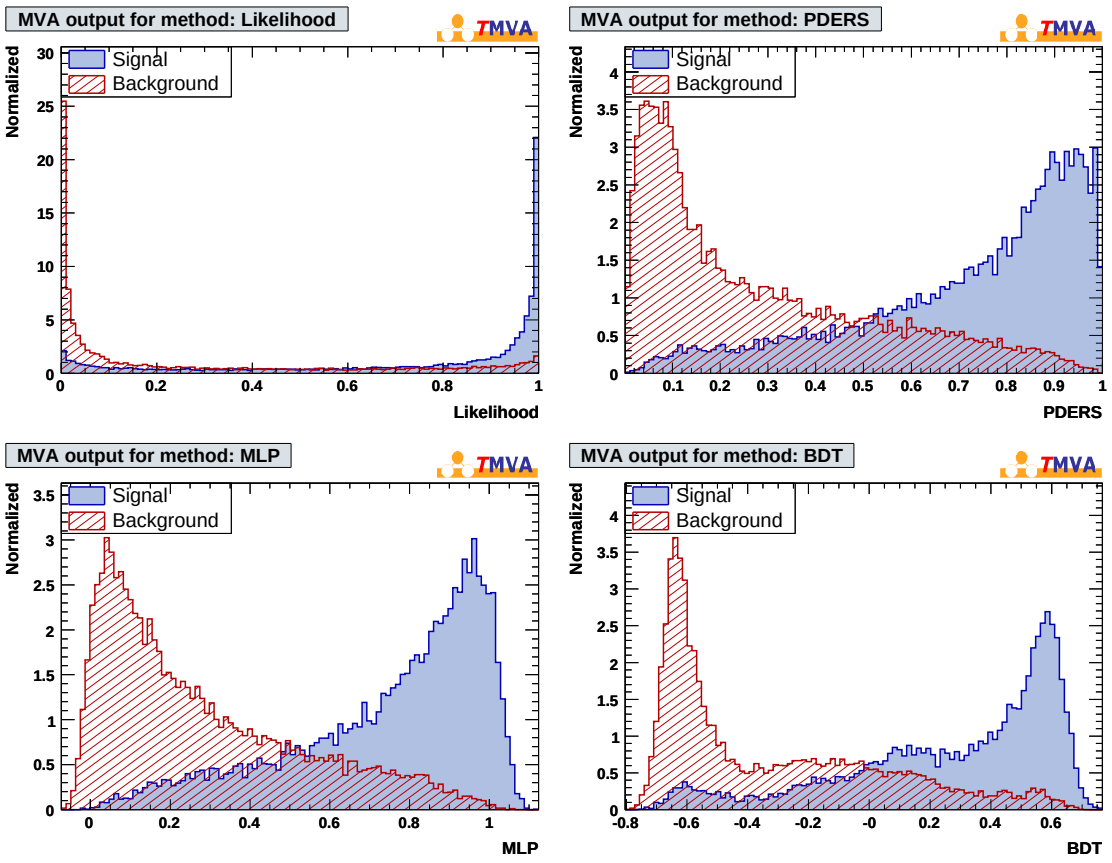


Figure 4: Example plots for classifier output distributions for signal and background events from the academic test sample. Shown are likelihood (upper left), PDE range search (upper right), MLP (lower left) and boosted decision trees.

```
TMVA::Factory* factory
          = new TMVA::Factory( "<JobName>", targetFile, "<options>" );
```

Code Example 5: Instantiating a Factory class object. The first argument is the user-defined job name which will also appear in the name of the weight files containing the training results. The second argument is the pointer to a writable `TFile` target file created by the user, where control and performance histograms are stored. Currently the only option that can be specified for the Factory is "V" for verbose print out.

### 3.1.1 Specifying training and testing data

The input data sets used for training and testing of the multivariate classifiers need to be handed to the Factory. TMVA supports ROOT `TTree` and derived `TChain` objects as well as text files. If ROOT trees are used, the signal and background events can be located in the same or in different trees. Overall weights can be specified for the signal and background training data (the treatment of event-by-event weights is discussed below).

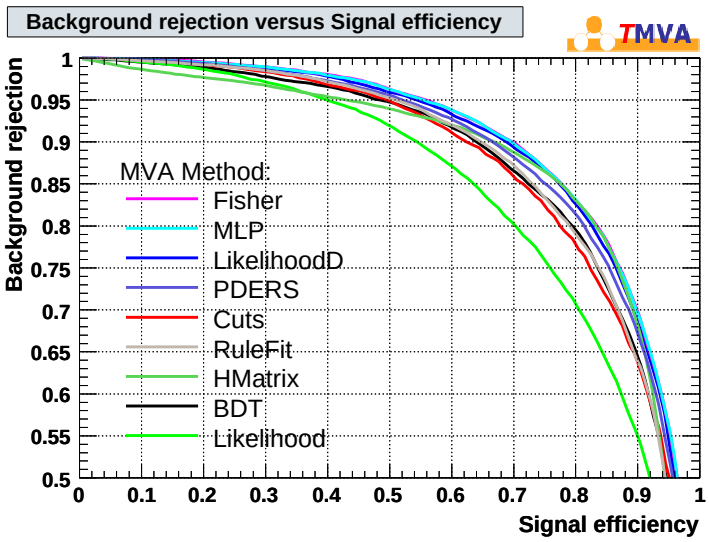


Figure 5: Example for the background rejection versus signal efficiency obtained for the various classifiers after evaluating the test sample.

Specifying training data in ROOT tree format with signal and background events being located in different trees:

```
TTree* sigTree = (TTree*)sigSource->Get( "<YourSignalTreeName>" );
TTree* bkgTree = (TTree*)bkgSource->Get( "<YourBackgrTreeName>" );

Double_t sigWeight = 1.0; // overall weight for all signal events
Double_t bkgWeight = 1.0; // overall weight for all background events

factory->SetInputTrees( sigTree, bkgTree, sigWeight, bkgWeight );
```

Code Example 6: Registration of signal and background ROOT trees read from `TFile` sources. Overall signal and background weights can also be specified. The `TTree` object may be replaced by a `TChain`.

Specifying training data in ROOT tree format with signal and background events being located in the same tree:

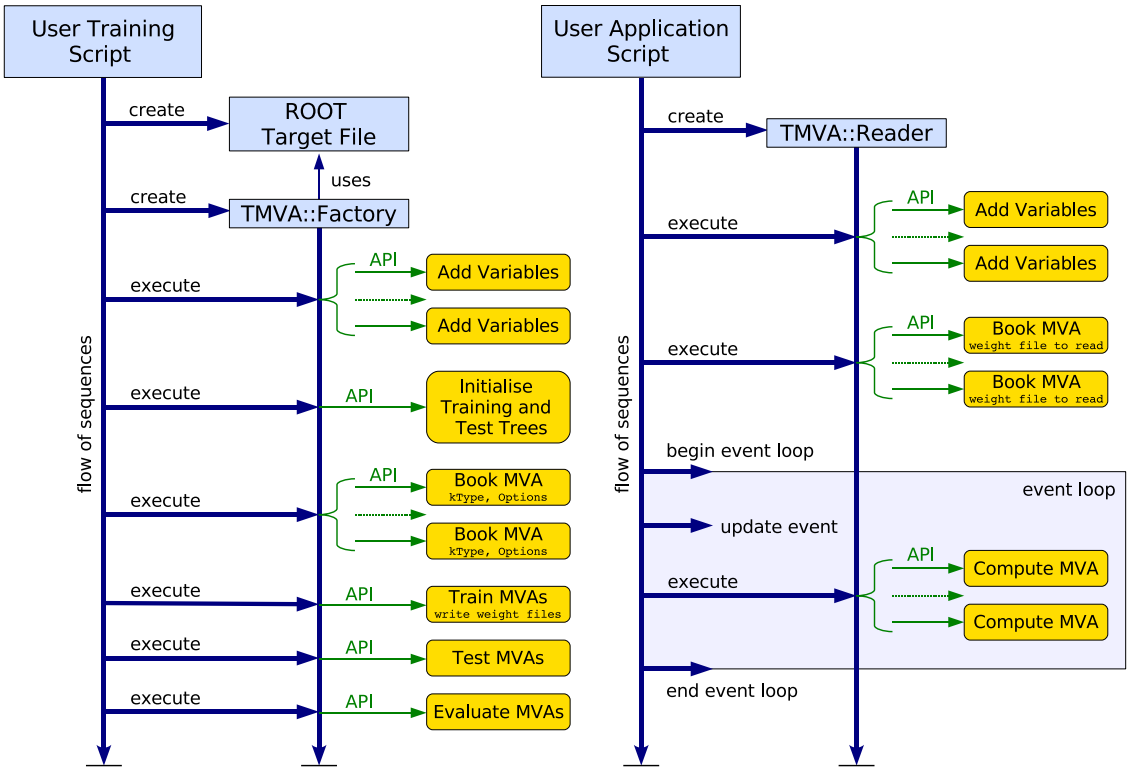


Figure 6: Left: sequence flow (top to bottom) of a typical TMVA training application. The user script can be a ROOT macro, C++ executable, python script or similar. The user creates a target ROOT `TFile`, which is used by the TMVA `Factory` object to write histograms and trees. The Factory, after creation by the user, organises the user's interaction with the TMVA modules. It is the only object directly created by the user. First the discriminating variables that must be `TFormula`-compliant functions of branches in the signal and background training tree are registered. Then, selected classifiers are booked through a type identifier, and configuration options are specified via an option string. The TMVA analysis proceeds by consecutively calling the training, testing and performance evaluation methods of the Factory. The training results for all classifiers used are written to custom weight files and the evaluation histograms are stored in the target file. They can be analysed with specific macros that come with TMVA (cf. Table 1). Right: sequence flow (top to bottom) of a typical TMVA analysis application. The classifiers that have been selected as appropriate in the preceding training and evaluation step are now used to classify data of unknown signal and background composition. First, a `Reader` class object is created, which serves as interface to the classifiers' response, just as was the Factory for the training and performance evaluation. The discriminating variables and references to locally declared memory placeholders are registered with the Reader. The variable names must coincide with those used for the training. The appropriate classifiers are booked through identification with their weight file, which fully configures the classifier. Only the bulk part of the name (that is the file name without extension) is given. The Reader adds the appropriate file extensions `.txt` and `.root` for the I/O operations. The user then runs the event loop, where for each event the values of the input variables are copied to the reserved memory addresses, and the MVA response values are computed. The user is responsible for the further use of the information.

```
TTree* inputTree = (TTree*)source->Get( "<YourTreeName>" );

TCut signalCut = ...;  // how to identify signal events in the TTree
TCut backgrCut = ...;  // how to identify backgr. events in the TTree

factory->SetInputTrees( inputTree, signalCut, backgrCut );
```

Code Example 7: Registration of a single ROOT tree containing the input data for signal *and* background, read from a `TFile` source. The `TTree` object may be replaced by a `TChain`. The cuts identify the event species.

Specifying training data in text format:

```
// Text file format (avaliable types: 'F' and 'I')
//   var1/F:var2/F:var3/F:var4/F
//   0.21293  -0.49200  -0.58425  -0.70591
//   ...
TString sigFile = "signal.txt";     // text file for signal
TString bkgFile = "background.txt"; // text file for background

Double_t sigWeight = 1.0; // overall weight for all signal events
Double_t bkgWeight = 1.0; // overall weight for all background events

factory->SetInputTrees( sigFile, bkgFile, sigWeight, bkgWeight );
```

Code Example 8: Registration of signal and background text files. Names and types of the input variables are given in the first line, followed by the values.

### 3.1.2 Selecting variables and variable transformations

The variables in input trees that are used to train the classifiers are registered with the Factory using the `AddVariable` method. It takes the variable name (string) and optionally its type (`'F'` and `'I'` are supported), which can be particularly useful for discrete variables (use `'I'`). If no type is given, `'F'` is used. The name must have a correspondence in the input ROOT tree or text file. It is also possible to specify variable expressions, just as for the `TTree::Draw` command (the expression is interpreted as a `TTreeFormula`).[5]

[5] There are restrictions in the use of array index specifications (true vector types in general): expressions like `"var1[0]"` are not permitted.

```
factory->AddVariable( "<YourVar1>",            'I' );
factory->AddVariable( "log(<YourVar2>)",       'F' );
factory->AddVariable( "<YourVar3>+<YourVar4>", 'F' );
```

Code Example 9: Declaration of variables used to train the classifiers. Each variable is specified by its name in the training tree (or text file), and optionally a type (`'F'` for float and `'I'` for integer, `'F'` is default). Here, `YourVar1` has discrete values and is thus declared as an integer. Just as in the `TTree::Draw` command, it is also possible to specify expressions of variables.

Individual events can be weighted, with the weights being a column or a function of columns of the input data sets. To tell TMVA to use these weights in the classifier training use the following command:

```
factory->SetWeightExpression( "<YourWeightExpression>" );
```

Code Example 10: Specification of individual weights applied to the training events. The expression must be a function of variables present in the input data set.

Some of the classifiers normalise the input variables so that $\min/\max(x_i) = 0/1$, $\forall i = 1, \ldots, n_{\text{var}}$. These are: Fisher discriminant, H-Matrix, the Clermont-Ferrand and MLP neural networks (customizable). In the present release of TMVA this option cannot be customized (excepting the MLP where the option `Normalize` exists), so that the user must be aware of it when interpreting the training results (weights).

### 3.1.3 Preparation of training and testing data

The input events that are handed to the Factory are internally copied and split into one training and one testing ROOT tree. This guarantees a statistically independent evaluation of the classifiers based on the test sample.[6] The numbers of events used in both samples are specified by the user. They must not exceed the entries of the input data sets. In case the user has provided a ROOT tree, the event copy is accelerated by disabling all branches not used by the input variables.

It is possible to apply selection requirements (cuts) to the input events. These requirements can depend on any variable present in the input data sets, i.e., they are not restricted to the variables used by the classifiers. The full command is as follows:

[6] A fully unbiased training and evaluation requires at least three statistically independent data sets. See comments in Footnote 9 on page 17.

```
TCut preselectionCut = "<YourSelectionString>";

factory->PrepareTrainingAndTestTree( preselectionCut,
                                     <NEventsTrain>, <NEventsTest> );
```

Code Example 11: Preparation of the internal training and testing trees. The signal and background events are split into ⟨`NEventsTrain`⟩ training events (equal number of both event species), and ⟨`NEventsTest`⟩ test events (unequal numbers possible). All selected events satisfy the preselection criteria. If ⟨`NEventsTest`⟩ is not specified, all remaining events are used after splitting off the training sample. For cases with very low statistics, the user may force TMVA to reuse the training events for testing with the command `factory->PrepareTrainingAndTestTree(preselectionCut,-1)`. In that case the evaluation may overstate the performance (cf. Sec. 3.1.8).

### 3.1.4 Booking the classifiers

All classifiers are booked via the Factory by specifying the classifier's type, plus a unique name given by the user, and a set of specific configuration options encoded in a string qualifier.[7] If the same type of classifier is booked several times with different options (which is useful to optimise the configuration of a classifier), the specified names should be different, allowing to separate the instances and their weight files. A booking example for the likelihood classifier is given in Code Example 12 below. Detailed descriptions of the configuration options are given in Secs. 4 and 5, and the default booking commands for all classifiers are given in Appendix A. With the classifier booking the initialization of the Factory is complete and no classifier-specific actions are left to do. The Factory takes care of the subsequent training, testing and evaluation of the classifiers.

```
factory->BookMethod( TMVA::Types::kLikelihood, "LikelihoodD",
       "!TransformOutput:Spline=2:NSmooth=5:Preprocess=Decorrelate" );
```

Code Example 12: Example booking of the likelihood classifier. The first argument is a unique type enumerator (the avaliable types can be looked up in `src/Types.h`), the second is a user-defined name which must be unique among all booked classifiers, and the third a configuration option string that is specific to the classifier. For options that are not set in the string default values are used. The syntax of the options should become clear from the above example. Individual options are separated by a ':'. Boolean variables can be set either explicitly as `MyBoolVar=True/False`, or just via `MyBoolVar/!MyBoolVar`. All specific options are explained in Secs. 4 and 5.

### 3.1.5 Training the classifiers

The training of the booked classifiers is invoked by the command

[7] In the TMVA package, a classifier is termed *Method*. According to that terminology, the Factory has a function `BookMethod`, and all *methods* are derived from the abstract classes `IMethod` and `MethodBase`.

```
factory->TrainAllMethods();
```

Code Example 13: Executing the classifier training via the Factory.

The training results are stored in the weight files which are saved in the directory `weights` (which, if not existing is created). The weight files are named `Jobname_ClassifierName.weights.⟨extension⟩`, where the job name has been specified at the instantiation of the Factory, and `ClassifierName` is the unique classifier name specified in the booking command. Each classifier writes two custom weight files, one in text format (extension is `txt`) where the classifier configuration options, controls and training weights are stored, and another in ROOT format (extension is `root`) containing `TObject`-derived objects (such as reference histograms for the likelihood classifier).

In addition to the classifier's output value $y_{\rm MVA}$, which is typically used to place a cut for the classification of an event as either signal or background, or which could be used in a subsequent likelihood fit, TMVA also provides signal and background PDFs, $\hat{y}_{S(B)}$. The PDFs can be used to derive classification probabilities for individual events. The techniques used to estimate the shapes of the PDFs are those developed for the likelihood classifier (see Sec. 5.2.2 for details) and can be customized individually for each method (the control options are given in Sec. 5). The probability for event $i$ to be of signal type is given by,

$$P_{\rm MVA}(i) = \frac{f_S \cdot \hat{y}_S(i)}{f_S \cdot \hat{y}_S(i) + (1 - f_S) \cdot \hat{y}_B(i)}\,, \tag{1}$$

where $f_S = N_S/(N_S + N_B)$ is the expected signal fraction, and $N_{S(B)}$ are the expected number of signal (background) events (default is $f_S = 0.5$).[8]

### 3.1.6 Testing the classifiers

The trained classifiers are applied to the testing data and provide scalar outputs according to which an event can be classified as either signal or background. The classifier outputs are stored in the test tree to which a column is added for each classifier. The tree is eventually written to the target file and can be directly analysed in a ROOT session. The testing of all booked classifiers is invoked by the command

```
factory->TestAllMethods();
```

Code Example 14: Executing the validation (testing) of the MVA classifiers via the Factory.

[8] The $P_{\rm MVA}$ distributions may exhibit a somewhat peculiar structure with frequent narrow peaks. They are generated by regions of classifier output values in which $\hat{y}_S \propto \hat{y}_B$ for which $P_{\rm MVA}$ becomes a constant.

### 3.1.7 Evaluating the classifiers

The Factory and data set classes of TMVA perform a preliminary assessment of the input variables used by the classifiers, such as computing linear correlation coefficients and ranking the variables according to their separation (see bullet below). The results are printed to standard output. After training and testing, also the linear correlation coefficients among the classifier outputs are printed. Moreover, overlap matrices are derived (and printed) for signal and background that determine the fractions of signal and background events that are equally classified by each pair of classifiers. This is useful when two classifiers have similar performance, but a significant fraction of non-overlapping events. In such a case a combination of the classifiers (e.g., in a *Committee* classifier) could improve the performance (this can be extended to any combination of any number of classifiers).

The performance evaluation in terms of efficiency, background rejection, etc., of the trained and tested classifiers is invoked by the command

```
factory->EvaluateAllMethods();
```

Code Example 15: Executing the performance evaluation via the Factory.

The optimal classifier to be used for a specific analysis strongly depends on the problem and no general recommendations can be given. To help with the choice TMVA computes a number of benchmark quantities that assess the performance of the classifiers on the independent test sample. These are

- The *signal efficiency at three representative background efficiencies* (the efficiency is equal to 1 − rejection) obtained from a cut on the classifier output. Also given is the area of the background rejection versus signal efficiency function (the larger the area the better the performance).
- The *separation* of a classifier $y$, defined by the integral [6]

$$\frac{1}{2}\int \frac{(\hat{y}_S(y) - \hat{y}_B(y))^2}{\hat{y}_S(y) + \hat{y}_B(y)} dy\,, \tag{2}$$

  where $\hat{y}_S$ and $\hat{y}_B$ are the signal and background PDFs of $y$, respectively. The separation is zero for identical signal and background shapes, and it is one for shapes with no overlap.
- The discrimination *significance* of a classifier, defined by the difference between the classifier means for signal and background divided by the quadratic sum of their root-mean-squares.
- The *average* $\int y\mu(\hat{y}_S(y))dy$ of the signal $\mu$-transform [7]. The $\mu$-transform of a classifier is the transformation that gives a uniform background distribution. In this way, the signal distribution of the $\mu$-transform can be directly compared among the

various classifiers. The stronger the peak towards one, the better is the discrimination. *This quantity has been removed from the standard evaluation output written by the Factory (for space reasons). It can be retrieved for each method through an appropriate accessor.*

The results of the evaluation are printed to standard output. In addition, smooth background rejection/efficiency versus signal efficiency curves (and the signal $\mu$-transform) are written to the target ROOT file, and can be plotted using custom macros (see Sec. 3.2).

### 3.1.8 Overtraining

Overtraining occurs when a data mining problem has too few degrees of freedom, because too many model parameters of a classifier were adjusted to too few data points. The sensitivity to overtraining therefore depends on the classifier. For example, a Fisher discriminant can hardly ever be overtrained, while, without the appropriate counter measures, boosted decision trees usually suffer from at least partial overtraining, owing to their large number of nodes. Overtraining leads to a seeming increase in the classification performance over the objectively achievable one, if measured on the training sample, and to an effective performance decrease when measured with an independent test sample. A convenient way to detect overtraining and to measure its impact is therefore to compare the classification results between training and test samples. Such a test is performed by TMVA with the results printed to standard output.

Various classifier-specific solutions to counteract overtraining exist. For example, binned likelihood reference distributions are smoothed before estimating their shapes, or unbinned kernel density estimators smear each training event before computing the PDF; neural networks steadily monitor the convergence of the error estimator between training and test samples[9] suspending the training when the test sample has passed its minimum; the number of nodes in boosted decision trees can be reduced by removing insignificant ones ("tree pruning"), etc.

## 3.2 ROOT macros for plotting training, testing and evaluation results

TMVA provides a simple GUI (`TMVAGui.C`, see Fig. 1), which interfaces ROOT macros that visualize the various steps of the training analysis. The macros are respectively located in `~/TMVA/macros/` (Sourceforge.net distribution) and `$ROOTSYS/tmva/test/` (ROOT distribution), and can also be executed from the command line. They are described in Table 1. All plots drawn are saved as png (or optionally eps, gif) files in the macro subdirectory `plots` which, if not existing, is created.

The binning and histogram boundaries for some of the histograms created throughout the

[9] Proper training and validation requires three statistically independent data sets: one for the parameter optimisation, another one for the overtraining detection, and the last one for the performance validation. In TMVA, the last two samples have been merged to increase statistics. The (usually insignificant) bias introduced by this on the evaluation results does not affect the analysis as far as cut efficiencies for the classifiers are independently validated with data.

training, testing and evaluation phases are controlled via the singleton class `TMVA::Config`. They can be modified as follows:

```
// To modify settings for the variable plotting, one can use the
// struct TMVA::Config::VariablePlotting
TMVA::gConfig().variablePlotting.timesRMS = 4.0;
TMVA::gConfig().variablePlotting.nbins1D  = 60;
TMVA::gConfig().variablePlotting.nbins2D  = 300;
```

Code Example 16: Modifying global parameter settings for the plotting of the discriminating input variables. The values given are the TMVA defaults.

## 3.3 The Reader

After training and evaluation, the most performing classifiers are selected and used to classify events in data samples with unknown signal and background composition. An example of how this *application phase* is carried out is given in `~/TMVA/macros/TMVApplication.C` (Sourceforge.net) and `$ROOTSYS/tmva/test/TMVApplication.C` (ROOT). Analogously to the Factory, the communication between the user application and the classifiers is interfaced by the TMVA *Reader*, which is created by the user:

```
TMVA::Reader* reader = new TMVA::Reader();
```

Code Example 17: Instantiating a Reader class object.

### 3.3.1 Specifying input variables

The user registers the names of the input variables with the Reader. They are required to be the same (and in the same order) as the names used for training (this requirement is not actually mandatory, but enforced to ensure the consistency between training and application). Together with the name is given the address of a local variable, which carries the updated input values during the event loop.

```
Float_t localVar1, localVar2, localVar3;

reader->AddVariable( "<YourVar1>",            &localVar1 );
reader->AddVariable( "log(<YourVar2>)",       &localVar2 );
reader->AddVariable( "<YourVar3>+<YourVar4>", &localVar3 );
```

Code Example 18: Declaration of the variables and references used as input to the classifiers. The order and naming of the variables must be consistent with the ones used for the training. The local variables are updated during the event loop, and through the references their values are known to the classifiers.

| Macro | Description |
|---|---|
| `variables.C` | Plots the signal and background MVA input variables (training sample). The second argument sets the preprocessing type (`type=0`, default, for no preprocessing, and `type=1,2` for decorrelated and PCA-transformed variables, cf. Sec. 4.1). |
| `correlationscatter.C` | Plots superimposed scatters and profiles for all pairs of input variables used during the training phase (separate plots for signal and background). As above, the second argument determines whether the original (`type=0`, default) or preprocessed (`type=1,2`) input variables are plotted. |
| `correlations.C` | Plots the linear correlation matrices for the signal and background training samples. |
| `mvas.C` | Plots the classifier response distributions of the test sample for signal and background. The second argument (`HistType=1`) allows to plot the probability distributions of the classifiers for an equal number of signal and background events (see Sec. 3.1.5). |
| `mvaeffs.C` | Signal and background efficiencies, obtained from cutting on the classifier outputs, versus the cut value. Also shown are the signal purity and the signal efficiency times signal purity assuming an equal number of signal and background events before cutting. |
| `efficiencies.C` | Background rejection (second argument `type=2`, default), or background efficiency (`type=1`), versus signal efficiency for the classifiers (test sample). The efficiencies are obtained by cutting on the classifier outputs. This is traditionally the best plot to assess the overall discrimination performance (ROC curve). |
| `likelihoodrefs.C` | Plots the reference PDFs of all input variables for the likelihood classifier and compares it to original distributions obtained from the training sample. |
| `network.C` | Draws the TMVA-MLP architecture including weights after training (does not work for the other ANNs). |
| `annconvergencetest.C` | Plots the MLP error-function convergence versus the training epoch for training and test events (does not work for the other ANNs). |
| `BDT.C(i)` | Draws the `i`th decision tree of the trained forest (default is `i=1`). The second argument is the weight file that contains the full architecture of the forest (default is `weights/MVAnalysis_BDT.weights.txt`). |
| `mvarefs.C` | Plots the classifier PDFs used to compute the probability response, and compares it to the original distributions. |

Table 1: List of available ROOT macros for the representation of the TMVA training and evaluation results. All macros take as first argument the name of the ROOT file containing the histograms (default is `TMVA.root`). The lower macros concern specific classifiers, and require that these classifiers have been included in the training.

### 3.3.2 Booking selected classifiers

The classifier(s) found to be most performing are booked with the Reader, using the weight files from the preceeding training job:

```
reader->BookMVA( "<YourClassifierName>", "<TheWeightFile.weights>" );
```

Code Example 19: Booking a multivariate classifier. The first argument is a user defined name to distinguish between classifiers (it does not need to be the same name as for training, although this could be a useful choice). The true type of the classifier and its full configuration are read from the weight file specified in the second argument.

### 3.3.3 Requesting the classifier output

Within the event loop, the response value $y_{\mathrm{MVA}}$ of a particular classifier for a given set of input variables (that are computed by the user) is obtained with the commands:

```
localVar1 = treeVar1;
localVar2 = TMath::Log(treeVar2);
localVar3 = treeVar3 + treeVar4;

Double_t mvaValue = reader->EvaluateMVA( "<YourClassifierName>" );
```

Code Example 20: Updating the local variables for an event, and obtaining the corresponding classifier output.

The classifier output $y_{\mathrm{MVA}}$ may then for example be cut on to increase the signal purity of the sample (the achievable purities can be read off the evaluation results obtained during the test phase), or it could enter a maximum-likelihood fit, etc.

The rectangular cut classifier is special since it returns a binary answer for a given set of input variables and cuts. The user must specify the desired signal efficiency to define the working point according to which the Reader will choose the cuts.

```
Bool_t passed = reader->EvaluateMVA( "Cuts", signalEfficiency );
```

Code Example 21: For the cut classifier, the second parameter gives the desired signal efficiency according to which the cuts are chosen. The return value is 1 for passed and 0 for retained.

Instead of the classifier response values, one may also retrieve the ratio (1) from the Reader, which, if properly normalised to the expected signal fraction in the sample, corresponds to a probability. The corresponding command is

```
Double_t pSig = reader->GetProba( "<YourClassifierName>", sigFraction );
```

Code Example 22: Requesting the event's signal probability from a classifier. The signal fraction is the parameter $f_S$ in Eq. (1).

### 3.4 Which classifier should I use for my problem?

There is obviously no common answer to that question. To guide the user, we have attempted an assessment of various relevant classifier properties in Table 1. Simplicity is a virtue, but only if it is not at the expense of discrimination power. Robustness with respect to overtraining could become an issue when the training sample is scarce. Some methods require more attention than others in this regard. For example, boosted decision trees are particularly vulnerable to overtraining if used without care. To circumvent overtraining a problem-specific adjustment of the pruning strength parameter is required. To assess whether a linear discriminant analysis (LDA) could be sufficient for a classification problem, the user is advised to analyse the correlations among the discriminating variables by inspecting scatter and profile plots (it is not enough to print the correlation coefficients, which by definition are linear only). Using an LDA greatly reduces the number of parameters to be adjusted and hence allow smaller training samples. For problems that require a high degree of optimisation and allow to form a large number of input variables, complex nonlinear methods like neural networks, boosted decision trees and RuleFit are more appropriate. For RuleFit we emphasize that the TMVA implementation differs from Friedman-Popescu's original code [19], with (yet) better robustness and out-of-the-box performance for the latter version. In particular, the behaviour of the original code with respect to nonlinear correlations and the curse of dimensionality would have merited two stars. We also point out that the excellent performance for by majority linearly correlated input variables is achieved somewhat artificially by adding a Fisher-like term to the RuleFit classifier (this is true for both implementations, cf. Sec. 5.8).

## 4 Data Preprocessing

A certain number of tools are centrally available in TMVA and can be accessed by all multivariate classifiers. For example, it is possible to preprocess the data prior to presenting it to the classifiers. Preprocessing can be useful to reduce correlations among the discriminating variables, to transform their shapes, or to accelerate the response time of a classifier.

### 4.1 Transforming input variables

Currently two preprocessing transformations are implemented in TMVA: decorrelation via the square-root of the covariance matrix and via a principal component decomposition. Technically, any transformation of the input variables is performed "on the fly" when the

| Criteria | | Classifiers | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Cuts | Likelihood | PDE-RS | H-Matrix | Fisher | ANN | BDT | RuleFit |
| Performance | None/linear correlations | ⋆ | ⋆⋆ | ⋆ | ⋆ | ⋆⋆ | ⋆⋆ | ⋆ | ⋆⋆ |
| | Nonlinear correlations | ◦ | ◦ | ⋆ | ◦ | ◦ | ⋆⋆ | ⋆⋆ | ⋆ |
| Speed | Training | ◦ | ⋆⋆ | ⋆⋆ | ⋆⋆ | ⋆⋆ | ⋆ | ◦ | ⋆ |
| | Response | ⋆⋆ | ⋆⋆ | ◦ | ⋆⋆ | ⋆⋆ | ⋆⋆ | ⋆ | ⋆ |
| Robustness | Overtraining | ⋆⋆ | ⋆ | ⋆ | ⋆⋆ | ⋆⋆ | ⋆ | ◦ | ⋆ |
| | Weak variables | ⋆⋆ | ⋆ | ◦ | ⋆⋆ | ⋆⋆ | ⋆ | ⋆⋆ | ⋆ |
| Curse of dimensionality | | ◦ | ⋆⋆ | ◦ | ⋆⋆ | ⋆⋆ | ⋆ | ⋆ | ⋆ |
| Transparency | | ⋆⋆ | ⋆⋆ | ⋆ | ⋆⋆ | ⋆⋆ | ◦ | ◦ | ◦ |

Table 1: Assessment of classifier properties. The symbols stand for the attributes "good" (⋆⋆), "OK" (⋆) and "bad" (◦). "Curse of dimensionality" refers to the "burden" of required increase in training statistics and processing time when adding more input variables. See also comments in text.

event is requested from the central `DataSet` class. Each classifier carries a variable transformation type together with a pointer to the object of its transformation class (all inherited from `VariableTransformBase`), which is owned by the `DataSet`. If no preprocessing is requested, an identity transform is applied. The `DataSet` registers the requested transformations and takes care not to recreate an identical transformation object (if requested) during the training phase. Hence if two classifiers wish to apply the same transformation, a single object is shared between them. Each classifier writes *its* transformation into its weight file once the training has converged. For testing and application of a classifier, the transformation is read from the weight file and a corresponding transformation object is created. Here each classifier owns its transformation so that no sharing of potentially different transformation objects occurs (they may have been obtained with different training data and/or under different conditions). A schematic view of the variable transformation interface used in TMVA is drawn in Fig. 7.

### 4.1.1 Variable decorrelation

A drawback of, for example, the projective likelihood classifier (see Sec. 5.2) is that it ignores correlations among the discriminating input variables. Because in most realistic use cases this is not an accurate conjecture it leads to performance loss. Also other classifiers, such as rectangular cuts or decision trees, and even multidimensional likelihood approaches underperform in presence of variable correlations.

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. The

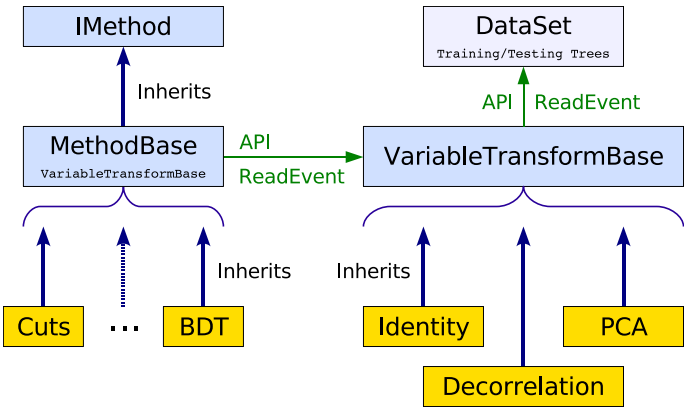


Figure 7: Schematic view of the variable transformation interface implemented in TMVA. Each concrete classifier derives from `MethodBase` (which is interfaced by `IMethod`), which holds a protected member object of type `VariableTransformBase`. The construction of the concrete variable transformation object proceeds in `MethodBase` according to the transformation method requested in the option string. The events used by the classifiers for training, testing and final classification analysis are read via an API of the `VariableTransformBase` class, which itself reads the events from the `DataSet`. The `DataSet` fills the current values into the reserved event addresses (the event content may either stem from the training or testing trees, or is set by the user's application via the `Reader` for the final classification analysis). The `VariableTransformBase` interface class ensures the proper transformation of all events seen by the classifiers.

square-root of a matrix $C$ is the matrix $C'$ that multiplied with itself yields $C$: $C = (C')^2$. TMVA computes the square-root matrix by means of diagonalising the (symmetric) covariance matrix

$$D = S^T C S \quad \Rightarrow \quad C' = S\sqrt{D}S^T \,, \tag{3}$$

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

The transformations are performed separately for signal and background events because their correlation patterns are usually different.[10] The decorrelation is complete only for linearly correlated and Gaussian distributed variables. In real-world use cases this is not often the case, so that sometimes only little additional information can be recovered by the decorrelation procedure. For highly non-linear problems the performance may even become worse with linear decorrelation. Non-linear classifiers without prior variable decorrelation should be used in such cases.

[10] Different transformations for signal and background events are only useful for methods that explicitly distinguish signal and background hypotheses. This is the case for the likelihood and PDERS classifiers. For all other methods the user must choose which transformation to use.

### 4.1.2 Principal component decomposition

Principal component decomposition or principal component analysis (PCA) as presently applied in TMVA is not very different from the above linear decorrelation. In common words, PCA 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 PCA-transformed coordinate system, the largest variance by any projection of the data comes to lie 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. This latter step however goes beyond straight variable transformation as performed in the preprocessing steps discussed here (it rather represents itself a full classifier). Hence all principal components are retained here.

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

$$x^{\rm PC}_{U,k}(i) = \sum_{\ell=1}^{n_{\rm var}} (x_{U,\ell}(i) - \overline{x}_{U,\ell})\, v^{(k)}_{U,\ell}\,, \qquad \forall k = 1, n_{\rm var}\,. \tag{4}$$

The tuples $\overline{\mathbf{x}}_U$ and $\mathbf{v}^{(k)}_U$ are the sample means and eigenvectors, respectively. They are computed by the ROOT class `TPrincipal`. The matrix of eigenvectors $V_U = (\mathbf{v}^{(1)}_U, \ldots, \mathbf{v}^{(n_{\rm var})}_U)$ obeys the relation

$$C_U \cdot V_U = D_U \cdot V_U\,, \tag{5}$$

where $C$ is the covariance matrix of the sample $U$, and $D_U$ is the tuple of eigenvalues. As for the preprocessing described in Sec. 4.1.1, the transformation (4) eliminates linear correlations for Gaussian variables.

## 4.2 Binary Search Trees

When frequent iterations over the training sample need to be performed, it is helpful to sort the sample before using it. Event sorting in *binary trees* is employed by the classifiers rectangular cut optimisation and PDERS.

Efficiently searching for and counting events that lie inside a multidimensional volume spanned by the discriminating input variables is accomplished with the use of a binary tree search algorithm [8].[11] It is realised in the class `BinarySearchTree`, which inherits

[11] The following is extracted from Ref. [9] for a two-dimensional range search example. Consider a random sequence of signal events $e_i(x_1, x_2)$, $i = 1, 2, \ldots$, which are to be stored in a binary tree. The first event in the sequence becomes by definition the topmost node of the tree. The second event $e_2(x_1, x_2)$ shall have a larger $x_1$-coordinate than the first event, therefore a new node is created for it and the node is attached to the first node as the right child (if the $x_1$-coordinate had been smaller, the node would have become the left child). Event $e_3$ shall have a larger $x_1$-coordinate than event $e_1$, it therefore should be attached to

from `BinaryTree`, and which is also employed to grow decision trees (cf. Sec. 5.7). The amount of computing time needed to sort $N$ events into the tree is [9] $\propto \sum_{i=1}^{N} \ln_2(i) = \ln_2(N!) \simeq N \ln_2 N$. Finding the events within the tree which lie in a given volume is done by comparing the bounds of the volume with the coordinates of the events in the tree. Searching the tree once requires a CPU time that is $\propto \ln_2 N$, compared to $\propto N^{n_{\rm var}}$ without prior event sorting.

# 5 The TMVA Classifiers

All TMVA classifying methods inherit from `MethodBase`, which implements basic functionality like the interpretation of common configuration options, the interaction with the training and test data sets, I/O operations and common performance evaluation calculus. The functionality each classifier is required to implement is defined in the abstract interface `IMethod`.[12] Each classifier must provide a function that creates a rank object (of type `Ranking`), which is an ordered list of the input variables prioritized according to criteria specific to that classifier.

If the option `CreateMVAPdfs` is set TMVA creates signal and background PDFs from the corresponding classifier response distributions using the training sample (cf. Sec. 3.1.5). The binning and smoothing properties of the underlying histograms can be customized via controls implemented in `MethodBase` (so that they are common to all classifiers). They are summarised in in Option Table 1.
The following sections describe the classifiers implemented in TMVA. For each classifier we proceed according to the following scheme: (*i*) a brief introduction, (*ii*) the description of the booking options required to configure the classifier, (*iii*) a description of the the classifier and TMVA implementation specifications, (*iv*) the properties of the variable ranking, and (*v*) a few comments on performance, favourable (and disfavoured) use cases, and comparisons with other classifiers.

## 5.1 Rectangular cut optimisation

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

the right branch below $e_1$. Since $e_2$ is already placed at that position, now the $x_2$-coordinates of $e_2$ and $e_3$ are compared, and, since $e_3$ has a larger $x_2$, $e_3$ becomes the right child of the node with event $e_2$. The tree is sequentially filled by taking every event and, while descending the tree, comparing its $x_1$ and $x_2$ coordinates with the events already in place. Whether $x_1$ or $x_2$ are used for the comparison depends on the level within the tree. On the first level, $x_1$ is used, on the second level $x_2$, on the third again $x_1$ and so on.

[12] Two constructors are implemented for each classifier: one that creates the classifier for a first time for training with a configuration ("option") string among the arguments, and another that recreates a classifier from an existing weight file. The use of the first constructor is demonstrated in the example macro `TMVAnalysis.C`, while the second one is employed by the Reader in `TMVApplication.C`. Other functions implemented by each classifier are: `Train` (called for training), `Write/ReadWeightsToStream` (I/O of specific training results), `WriteMonitoringHistosToFile` (additional specific information for monitoring purposes) and `CreateRanking` (variable ranking).

| Option | Values | Description |
|---|---|---|
| `VarTransform` | `None*, Decorrelate, PCA` | Transformation method for input variables |
| `V` | `False` | Verbose flag |
| `H` | `False` | Help flag |
| `NbinsMVAPdf` | 60 | Number of bins in histograms used to build the classifier PDFs |
| `NsmoothMVAPdf` | 2 | Number of smoothing iterations |

Option Table 1: Configuration options common to all classifiers (but which can be controlled individually for each classifier). Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The lower options in the table control the PDF fitting of the classifiers.

on discriminating variables. Unlike all other classifiers in TMVA, the cut classifier only returns a binary response (signal *or* background). The optimisation of cuts performed by TMVA maximises the background rejection at given signal efficiency, and scans over the full range of the latter quantity. Dedicated analysis optimisation for which, e.g., the signal *significance* is maximised requires the expected signal and background yields to be known before applying the cuts. This is not the case for a multi-purpose discrimination and hence not used by TMVA. However, the cut ensemble leading to maximum significance corresponds to a particular working point on the efficiency curve, and can hence be easily derived after the cut optimisation scan has converged.[13]

TMVA implements three methods for cut optimisation: Monte Carlo (MC) sampling, a Genetic Algorithm (GA), and Simulated Annealing (SA, which is however depreciated for the present release, since it has not yet been sufficiently validated).[14] Attempts to use MINUIT (Simplex or Migrad) have not shown satisfactory results, with frequently failing fits because of the non-unique and non-global solution space. For most applications GA should be the most performing optimisation method.

[13] Assuming a large enough number of events so that Gaussian statistics is applicable, the significance for a signal is given by $\mathcal{S} = \varepsilon_{\mathcal{S}}\mathcal{N}_{\mathcal{S}}/\sqrt{\varepsilon_{\mathcal{S}}\mathcal{N}_{\mathcal{S}} + \varepsilon_{\mathcal{B}}(\varepsilon_{\mathcal{S}})\mathcal{N}_{\mathcal{S}}}$, where $\varepsilon_{S(B)}$ and $N_{S(B)}$ are the signal and background efficiencies for a cut ensemble and the event yields before applying the cuts, respectively. The background efficiency $\varepsilon_B$ is expressed as a function of $\varepsilon_S$ using the TMVA evaluation curve obtained form the test data sample. The maximum significance is then found at the root of the derivative

$$\frac{d\mathcal{S}}{d\varepsilon_S} = N_S \frac{2\varepsilon_B(\varepsilon_S)N_B + \varepsilon_S\left(N_S - \frac{d\varepsilon_B(\varepsilon_S)}{d\varepsilon_S}N_B\right)}{2\left(\varepsilon_S N_S + \varepsilon_B(\varepsilon_S)N_B\right)^{3/2}} = 0\,, \tag{6}$$

which depends on the problem.

[14] We note that cut optimisation is not a genuine multivariate analyser method, because no combination of the variables is achieved. Neither does a cut on one variable depend on the value of another variable (like it is the case for Decision Trees), nor can a, say, background-like value of one variable in a signal event be counterweighed by signal-like values of the other variables (like it is the case for the likelihood method).

The training events are sorted in *binary trees* prior to the optimisation, which significantly reduces the computing time required to determine the number of events passing a given cut ensemble (cf. Sec. 4.2).

### 5.1.1 Booking options

The rectangular cut optimisation is booked through the Factory via the command:

```
factory->BookMethod( Types::kCuts, "Cuts", "<options>" );
```

Code Example 23: Booking of the cut optimisation classifier: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

The configuration options for the various cut optimisation techniques are given in Option Table 2.

### 5.1.2 Description of the classifier and its implementation

The cut optimisation analysis proceeds by first building binary search trees for signal and background. For each variable, statistical properties like mean, root-mean-squared (RMS), variable ranges are computed to guide the search for optimal cuts. Cut optimisation requires an estimator that quantifies the goodness of a given cut ensemble. Maximising this estimator minimises (maximises) the background efficiency, $\varepsilon_B$ (background rejection, $r_B = 1 - \varepsilon_B$) for each signal efficiency $\varepsilon_S$.

All three optimisation methods act on the assumption that one minimum and one maximum requirement on each variable is sufficient to optimally discriminate signal from background (i.e., the signal is clustered). If this is not the case, the variables must be transformed prior to the cut optimisation 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. The resulting efficiencies are therefore rational numbers that may exhibit visible discontinuities when the number of training events is small and an efficiency is either very small or very large. Another way to compute efficiencies is to parametrise the probability density functions of all input variables and to thus achieve continuous efficiencies for any cut value. Note however that this method expects the input variables to be uncorrelated! Nonvanishing correlations would lead to incorrect efficiency estimates and hence to underperforming cuts. The two methods are chosen with the option `EffMethod` set to `EffSel` and `EffPDF`, respectively.

| Option | Values | Description |
|---|---|---|
| Method | MC, GA, SA | Optimisation method |
| EffMethod | EffSel, EffPDF | Selection method |
| MC_NRandCuts | 100000 | Monte Carlo sample size |
| MC_AllVarProp | AllNotEnforced, AllF(Max, Min, Smart) | Variable properties that can be used to inject prior information on cut boundaries; this allows to reduce the number of sampled cuts |
| MC_Var[i]Prop | (as above) | Same as above but per variable |
| GA_nsteps | 30 | Stop if the fitness did not increase by at least GA_convCrit during the previous GA_nsteps generations |
| GA_convCrit | 0.0001 | see GA_nsteps |
| GA_cycles | 3 | Number of optimisation cycles |
| GA_popSize | 100 | Number of individuals in population |
| GA_SC_steps | 10 | If an improvement occurred in GA_SC_rate steps of a period of the last GA_SC_steps steps, a cut-variation factor is multiplied by GA_SC_factor |
| GA_SC_rate | 5 | see GA_SC_steps |
| GA_SC_factor | 0.95 | see GA_SC_steps |
| SA_MaxCalls | 5000000 | Max. number of minimisation calls |
| SA_TemperatureGradient | 0.7 | Temperature gradient |
| SA_UseAdaptiveTemperature | True | Use adaptive temperature |
| SA_InitialTemperature | 100000 | Initial temperature |
| SA_MinTemperature | 500 | Minimum temperature |
| SA_NFunLoops | 5 | Number of function loops |
| SA_Eps | 1.0e-04 | Minimum improvement in previous step required to continue annealing |
| SA_NEps | 4 | Number of functions to satisfy SA_Eps |

Option Table 2: Configuration options for cut optimisation. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The options in Option Table 1 can also be configured. The prefixes MC (GA, SA) denote options that set properties of the Monte Carlo (Genetic Algorithm, Simulated Annealing) optimisation procedure (selected by the option Method).

### Monte Carlo sampling

The simplest and most straightforward, albeit somewhat inefficient method to solve the optimisation problem is to randomly sample the minimum and maximum requirements for all discriminating variables. Each sample corresponds to a point in the $(\varepsilon_S, r_B)$ plane. The $\varepsilon_S$ dimension is (finely) binned and a cut sample is retained if its $r_B$ value is larger than the value already contained in that bin. This way a reasonably smooth efficiency curve can be obtained if the number of input variables is not too large (the required number of MC samples grows with powers of $2n_{\text{var}}$).

Prior information on the variable distributions can be used to reduce the number of cuts that need to be sampled. For example, if a discriminating variable follows Gaussian distributions for signal and background, with equal width but a larger mean value for the background distribution, there is no useful minimum requirement (other than $-\infty$) so that a single maximum requirement is sufficient for this variable. To instruct TMVA to remove obsolete requirements, the option `MC_Var[i]Prop` must be used, where `[i]` indicates the counter of the variable (following the order in which they have been registered with the Factory, beginning with 0) must be set to either `FMax` or `FMin`. TMVA is capable of automatically detecting which of the requirements should be removed. Use the option `MC_Var[i]Prop=FSmart` (where again `[i]` must be replaced by the appropriate variable counter, beginning with 0). Note that in many realistic use cases the mean values between signal and background of a variable are indeed distinct, but the background can have large tails. In such a case, the removal of a requirement is inappropriate, and would lead to underperforming cuts.

### Genetic Algorithm

Genetic Algorithm is a technique to find approximate solutions to optimisation or search problems. The problem is modeled by a group (*population*) of abstract representations (*genomes*) of possible solutions (*individuals*). By applying means similar to processes found in biological evolution the individuals of the population should evolve towards an optimal solution of the problem. Processes which are usually modeled in evolutionary algorithms — of which Genetic Algorithms are a subtype — are inheritance, mutation and "sexual recombination" (also termed *crossover*).

Apart from the abstract representation of the solution domain, a *fitness* function must be defined. Its purpose is the evaluation of the goodness of an individual. The fitness function is problem dependent. In cut optimisation for instance, the quality of a rectangular cut is given by good background rejection combined with high signal efficiency. It either returns a value representing the individual's goodness or it compares two individuals and indicates which of them performs better.

The Genetic Algorithm proceeds as follows:

- *Initialization*: A starting population is created. Its size depends on the problem to be solved. Each individual belonging to the population is created by randomly setting

the parameters (cuts) of the abstract representation (variables), thus producing a point (cut ensemble) in the solution domain of the initial problem.

- *Evaluation*: Each individual is evaluated using the fitness function.
- *Selection*: Individuals are kept or discarded as a function of their fitness. Several selection procedures are possible. The simplest one is to separate out the worst performing fraction of the population. Another possibility is to decide on the individual's survival by assigning probabilities that depend on the individual's performance compared to the others.
- *Reproduction*: The surviving individuals are copied, mutated and crossed-over until the initial population size is reached again.
- *Termination*: The evaluation, selection and reproduction steps are repeated until a maximum number of cycles is reached or an individual satisfies a maximum-fitness criterion. The best individual is selected and taken as solution to the problem.

The TMVA Genetic Algorithm implementation provides a certain number of controls that can be set through the options (cf. Table 2). The parameter `GA_popSize` determines the number of individuals (cut ensembles) which are created at each generation of the Genetic Algorithm. At the initialization, all parameters of all individuals are chosen randomly. The individuals are evaluated in terms of their background rejection and signal efficiency. Each cut ensemble giving an improvement in the background rejection for a specific signal efficiency bin is immediately stored. Each individual's fitness is assessed, where the fitness is largely determined by the difference of the best found background rejection for a particular bin of signal efficiency and the value produced by the current individual. The same individual that has at one generation a very good fitness will have only average fitness at the following generation. This forces the algorithm to focus on the region where the potential of improvement is the highest.

Individuals with a good fitness are selected to produce the next generation. The new individuals are created by crossover and mutated afterwards. Mutation changes some values of some parameters of some individuals randomly following a Gaussian distribution function. The width of the Gaussian can be altered by the parameter `GA_SC_factor`. The current width is multiplied by this factor when within the last `GA_SC_steps` generations more than `GA_SC_rate` improvements have been obtained. If there were `GA_SC_rate` improvements the width remains unchanged. Were there, on the other hand, less than `GA_SC_rate` improvements, the width is divided by `GA_SC_factor`. This allows to influence the speed of searching through the solution domain.

The cycle of evaluating the fitness of the individuals of a generation and producing a new generation is repeated until the improvement within the last `GA_nsteps` has been less than `GA_convCrit` in terms of background rejection. The minimisation is then considered to have converged. The whole cycle from initialization over evaluation of fitness, selection, reproduction and determining the improvement is repeated `GA_cycles` times. The Genetic Algorithm has finished.

#### Simulated Annealing

As the Genetic Algorithm, Simulated Annealing attempts to solve a minimisation problem with manifold discrete or continuous, local or global solutions. For example, when slowly cooling down ("annealing") a metal its atoms move towards a state of lowest energy, while for fast annealing the atoms tend to freeze in intermediate higher energy states. For infinitesimal annealing activity the system will always converge in its global energy minimum [10].

This physical principle can be simulated to achieve slow, but correct convergence of an optimisation problem with multiple solutions. Recovery out of local minima is achieved by assigning the probability [11]

$$p(\Delta E) \propto \exp\left(-\frac{\Delta E}{T}\right) , \tag{7}$$

to a perturbation of the parameters (the cuts) leading to a shift $\Delta E$ in the energy of the system. The probability of such perturbations to occur decreases with the size of a positive energy coefficient of the perturbation, and with the ambient temperature ($T$). The TMVA implementation of Simulated Annealing uses adaptive adjustment of the perturbation and temperature gradients.

*Although the Simulated Annealing algorithm is technically functional, it has not yet been optimised so that its use is depreciated until further notice.*

### 5.1.3 Ranking

The present implementation of Cuts does not provide a ranking of the input variables.

### 5.1.4 Performance

The Genetic Algorithm currently provides the best cut optimisation convergence. However, it is found that with rising number of discriminating input variables the goodness of the solution found (and hence the smoothness of the background-rejections versus signal efficiency plot) deteriorates quickly. Rectangular cut optimisation should therefore be reduced to the variables that have the largest discriminating power.

If variables with excellent signal from background separation exist, applying cuts can be quite competitive with more involved classifiers. Cuts are known to underperform in presence of strong non-linear correlations and/or if several weakly discriminating variables are used. In the latter case, a true multivariate combination of the information will be rewarding.

## 5.2 Projective likelihood estimator (PDE approach)

The method of maximum likelihood consists of building a model out of probability density functions (PDFs) that reproduces the input variables for 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, and normalising this by the sum of the signal and background likelihoods. Correlations among the variables are ignored.

### 5.2.1 Booking options

The likelihood classifier is booked via the command:

```
factory->BookMethod( Types::kLikelihood, "Likelihood", "<options>" );
```

Code Example 24: Booking of the (projective) likelihood classifier: the first argument is the predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

The likelihood configuration options are given in Option Table 3.

| Option | Values | Description |
|---|---|---|
| `Spline` | `0, 1, 2*, 3, 5` | Degree of splines used to interpolate the reference histograms |
| `NSmooth` | `5` | Number of smoothing iterations for the input histograms |
| `NAvEvtPerBin` | `25` | Average number of events per bin in each reference histogram (to allow an adaptive number of bins) |
| `UseKDE` | `False` | Use kernel density estimator (KDE) instead of spline functions |
| `KDEtype` | `Gauss*` | KDE kernel type (currently only Gauss) |
| `KDEiter` | `Nonadaptive*, Adaptive` | Nonadaptive or adaptive number of iterations (see text) |
| `KDEFineFactor` | `1` | Finetuning factor for the adaptive KDE |
| `KDEborder` | `None*, Renorm, Mirror` | Method for correcting boundary/border effects |
| `TransformOutput` | `False` | Transform likelihood output by inverse sigmoid function |

Option Table 3: Likelihood configuration options. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The options in Option Table 1 can also be configured.

### 5.2.2 Description of the classifier and its implementation

The likelihood ratio $y_{\mathcal{L}}(i)$ for event $i$ is defined by

$$y_{\mathcal{L}}(i) = \frac{\mathcal{L}_S(i)}{\mathcal{L}_S(i) + \mathcal{L}_B(i)}\,, \tag{8}$$

where

$$\mathcal{L}_{S(B)}(i) = \prod_{k=1}^{n_{\rm var}} p_{S(B),k}(x_k(i))\,, \tag{9}$$

and where $p_{S(B),k}$ is the signal (background) PDF for the $k$th input variable $x_k$. The PDFs are normalised

$$\int_{-\infty}^{+\infty} p_{S(B),k}(x_k)dx_k = 1\,, \quad \forall k. \tag{10}$$

It can be shown that in absence of model inaccuracies (such as correlations between input variables not removed by the decorrelation procedure, or an inaccurate probability density model), the ratio (8) provides optimal signal from background separation for the given set of input variables.

Since the parametric form of the PDFs is generally unknown, the PDF shapes are empirically approximated from the training data by nonparametric functions, which can be polynomial splines of various degrees fitted to histograms or unbinned kernel density estimators (KDE), as discussed below.

A certain number of primary validations are performed during the PDF creation, the results of which are printed to standard output. Among these are the computation of a $\chi^2$ estimator between all nonzero bins of the original histogram and its PDF, and a comparison of the number of outliers (in sigmas) found in the original histogram with respect to the (smoothed) PDF shape, with the statistically expected one. The fidelity of the PDF estimate can be also inspected visually by executing the macro `likelihoodrefs.C` (cf. Table 1).

#### Nonparametric PDF parameterisation using spline functions

Polynomial splines are fitted to binned histograms according to the following procedure.

1. For each input variable, a histogram is filled with the training data. The upper and lower bounds of the histogram coincide with the limits found in the data (or they are equal to [0,1] if the input variables are normalised). The (equidistant) binning is chosen so that the average number of entries per bin corresponds to the number (`NAvEvtPerBin`) defined in the option string.

2. The histogram is smoothed `NSmooth` times using the method `TH1::SmoothArray(.)`, which is an implementation of the algorithm 353QH twice [12]. The appropriate number of smoothing iterations depends on the shape of the histogram. Since

smoothing tends to even out all structures from the histogram, narrow structures (e.g., peaks) support less smoothing than broad ones.

3. The smoothed histogram is used to construct an object of the class `PDF`, where it is cloned and the bins are fit to polynomial interpolation functions, (splines – derivatives of the ROOT class `TSpline`). The available splines are: degree 0 (the original histogram is kept), which is useful for discrete variables; degree 1 (linear), 2 (quadratic), 3 (cubic) and degree 5. Splines of degree two or above render the PDF continuous and differentiable in all points excluding the interval borders, which in turn ensures the same property for the likelihood ratio (8). Since cubic (and higher) splines equalize the first and second derivatives at the spline transitions, the resulting curves, although mathematically smooth, can wiggle in quite unexpected ways. Furthermore, there is no local control of the spline: moving one control point (bin) causes the entire curve to change, not just the part near the control point. To ensure a safe interpolation, quadratic splines are used by default.

4. To speed up the numerical access to the probability densities, the spline functions are stored into a finely binned ($10^4$ bins) histogram, where adjacent bins are interpolated by a linear function. Only after this step, the PDF is normalised according to (10).

#### Nonparametric PDF parameterisation using kernel density estimators

Another type of nonparametric approximation of the PDFs is achieved with kernel density estimators (KDE). As opposed to splines KDEs are obtained from unbinned data. The idea of the approach is to estimate the shape of a PDF by the sum over *smeared* training events. One then finds for a PDF $p(x)$ of a variable $x$ [13]

$$p(x) = \frac{1}{N\,h}\sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right) = \frac{1}{N}\sum_{i=1}^{N} K_h(x - x_i)\ , \tag{11}$$

where $N$ is the number of training events, $K_h(t) = K(t/h)/h$ is the kernel function, and $h$ is the *bandwidth* of the kernel (also termed the *smoothing parameter*). Currently, only a Gaussian form of $K$ is implemented, where the exact form of the kernel function is of minor relevance for the quality of the shape estimation. More important is the choice of the bandwidth.

The KDE smoothing can be applied in either nonadaptive (NA) or adaptive form (A), the choice of which is controlled by the option `KDEiter`. In the nonadaptive case the bandwidth $h_{\rm NA}$ is kept constant for the entire training sample. As optimal bandwidth can be taken the one that minimizes the *asymptotic mean integrated square error* (AMISE). For the case of a Gaussian kernel function this leads to [13]

$$h_{\rm NA} = \left(\frac{4}{3}\right)^{1/5} \sigma_x N^{-1/5}\,, \tag{12}$$

where $\sigma_x$ is the RMS of the variable $x$.

The so called *sample point adaptive* method uses as input the result of the nonadaptive KDE, but also takes into account the local event density. The adaptive bandwidth $h_{\rm A}$ then becomes a function of $p(x)$ [13]

$$h_{\rm A}(x) = \frac{h_{\rm NA}}{\sqrt{p(x)}}\,. \tag{13}$$

The adaptive approach improves the shape estimation in regions with low event density. On the contrary, in regions with high event density it can give rise to "over-smoothing" of fine structures such as narrow peaks. The degree of smoothing can be tuned by multiplying the bandwidth $h_{\rm A}(x)$ with the user-specified factor `KDEFineFactor`.

For practical reasons, the KDE implementation in TMVA differs somewhat form the procedure described above. Instead of using the unbinned training data, finely-binned histograms are used as inputs, which allows to speed up the algorithm. In a second step, a `KDEKernel` class object is created where the calculation of the bandwidth $h_{\rm NA}$ is performed. If the algorithm is run in the adaptive mode the nonadaptive step is also performed and the output is used to compute $h_{\rm A}(x)$ for the adaptive part. In a third step, a smoothed histogram estimating the PDF shape is filled by looping over the binned input histogram and summing up the kernel functions. Here $h_{\rm NA}$ is used for the nonadaptive mode and $h_{\rm A}(x)$ for the adaptive mode. Finally, the smoothed histogram is used to construct a `PDF` class object.

Both the nonadaptive and the adaptive methods can suffer from the so-called *boundary problem*. It occurs for instance if the original distribution is nonzero below a physical boundary value and zero above. This property cannot be reproduced by the KDE procedure. In general, the stronger the discontinuity the more acute is the boundary problem. TMVA provides three options under the term `KDEborder` that allow to treat boundary problems.

- `KDEborder=None`
  No boundary treatment is performed. The consequence is that close to the boundary the KDE result will be inaccurate: below the boundary it will underestimate the PDF while it will not drop to zero above. In TMVA the PDF resulting from KDE is in fact a (finely-binned) histogram, with bounds equal to the minimum and the maximum values of the original distribution. Hence, the boundary value will be at the edge of the PDF (histogram), and a drop of the PDF due to the closeness of the boundary can be observed (while the artificial enhancement beyond the boundary will fall outside of the histogram). In other words, for training events that are close to the boundary some fraction of the probability "flows" outside the histogram (probability *leakage*). As a consequence, the integral of the kernel function inside the histogram borders is smaller than one.

- `KDEborder=Renorm`
  The probability leakage is compensated by renormalising of the kernel function so that the integral inside the histogram borders is equal to one.

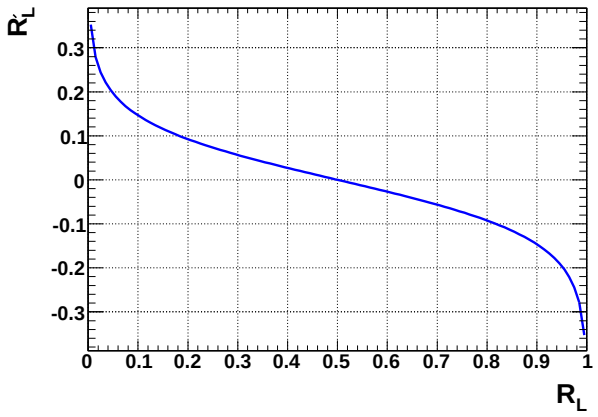


Figure 8: Transformation (14) of the likelihood output.

- `KDEborder=Mirror`
  The original distribution is "mirrored" around the boundary. To the events originating from this mirror copy the same procedure is applied as for the original ones: each of them is smeared by a kernel function and its contribution inside the histogram (PDF) boundaries is added to the PDF. The mirror copy exactly compensates the probability leakage.

#### Transforming the likelihood output

If a data-mining problem offers a large number of input variables, or variables with excellent separation power, the likelihood response $y_{\mathcal{L}}$ is often strongly peaked at 0 (background) and 1 (signal). Such a response is inconvenient for the use in subsequent analysis steps. TMVA therefore allows to transform the likelihood output by an inverse sigmoid function that zooms into the peaks

$$y_{\mathcal{L}}(i) \longrightarrow y'_{\mathcal{L}}(i) = -\tau^{-1} \ln\left(y_{\mathcal{L}}^{-1} - 1\right) , \tag{14}$$

where $\tau = 15$ is used (hardcoded). Note that $y'_{\mathcal{L}}(i)$ is no longer contained within $[0, 1]$ (see Fig. 8). The transformation (14) is enabled (disabled) with the likelihood booking option `TransformOutput=True(False)`.

### 5.2.3 Ranking

The present likelihood implementation does not provide a ranking of the input variables.

### 5.2.4 Performance

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. The neglect of correlations between input variables in the model (9), often leads to a diminution of the discrimination performance. While linear Gaussian correlations can be rotated away (see Sec. 4.1), such an ideal situation is rarely given. Positive correlations lead to peaks at both $y_{\mathcal{L}} \to 0, 1$. Correlations can be reduced by categorizing the data samples and building an independent likelihood classifier for each event category. Such categories could be geometrical regions in the detector, kinematic properties, etc. In spite of this, realistic applications with a large number of input variables are often plagued by irreducible correlations, so that projective likelihood approaches like the one discussed here are underperforming. This finding lead to the development of the many alternative classifiers that exist in statistical theory today.

## 5.3 Multidimensional likelihood estimator (PDE range-search approach)

This is a generalization of the projective likelihood classifier described in Sec. 5.2 to $n_{\text{var}}$ dimensions, where $n_{\text{var}}$ is the number of input variables used. If the multidimensional PDF for signal and background were 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.[15] 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 *PDERS*, has been suggested in Ref. [9]. The PDE for a given test event (discriminant) is obtained by counting the (normalised) number of signal and background (training) events that occur in the "vicinity" of the test event. The classification of the test event may then be conducted on the basis of the majority of the nearest training events. The $n_{\text{var}}$-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. PDERS is a variant of a K nearest neighbour algorithm.

### 5.3.1 Booking options

The PDERS classifier is booked via the command:

```
factory->BookMethod( Types::kPDERS, "PDERS", "<options>" );
```

Code Example 25: Booking of PDERS: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

[15] Due to correlations between the input variables, only a sub-space of the full phase space may be populated.

The configuration options for the PDERS classifier are given in Option Table 4.

| Option | Values | Description |
|---|---|---|
| `VolumeRangeMode` | `Unscaled, RMS, MinMax, Adaptive*` | Definition of the volume reference |
| `DeltaFrac` | `3.0` | Volume size: multiplies MinMax or RMS |
| `NEventsMin` | `100` | Minimum number of events required in adaptive volume |
| `NEventsMax` | `200` | Maximum number of events required in adaptive volume |
| `MaxVIterations` | `50` | Maximum number of iterations for the adaptive volume search |
| `InitialScale` | `0.99` | Initial size of adaptive volume (compared to full volume spanned by data) |
| `KernelEstimator` | `Box*, Sphere, Teepee, Gauss, Sinc3(5,7,9,11), Lanczos2(3,5,8)` | Kernel estimator function |
| `GaussSigma` | `0.2` | Width (with respect to volume size) of Gaussian kernel estimator |

Option Table 4: PDERS configuration options. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The options in Option Table 1 can also be configured.

### 5.3.2 Description of the classifier and its implementation

To classify an event as being either of signal or of background type, a *local* estimate of the probability density of it belonging to either class is computed. The method of PDERS provides such an estimate by defining a volume ($V$) around the test event ($i$), and by counting the number of signal ($n_S(i,V)$) and background events ($n_B(i,V)$) obtained from the training sample in that volume. The ratio

$$y_{\mathrm{PDERS}}(i,V) = \frac{1}{1+r(i,V)} \tag{15}$$

is taken as the estimate, where $r(i,V) = (n_B(i,V)/N_B) \cdot (N_S/n_S(i,V))$, and $N_{S(B)}$ is the total number of signal (background) events in the training sample. The estimator $y_{\mathrm{PDERS}}(i,V)$ peaks at 1 (0) for signal (background) events. The counting method averages over the PDF within $V$, and hence ignores the available shape information inside (and outside) that volume.

### Binary tree search

Efficiently searching for and counting the events that lie inside the volume is accomplished with the use of a $n_{\text{var}}$-variable binary tree search algorithm [8] (cf. Sec. 4.2).

### Choosing a volume

The TMVA implementation of PDERS optionally provides four different volume definitions:

- `Unscaled`
  The simplest volume definition. The user specifies a rigid box of size `DeltaFrac` in units of the variables. This method was the one originally used by the developers of PDERS [9].

- `MinMax`
  The volume is defined in each dimension (i.e., input variable) with respect to the full range of values found for that dimension in the training sample. The fraction of this volume used for the range search is defined by the option `DeltaFrac`.

- `RMS`
  The volume is defined in each dimension with respect to the RMS of that dimension (input variable), estimated from the training sample. The fraction of this volume used for the range search is defined by the option `DeltaFrac`.

- `Adaptive`
  A volume is defined in each dimension with respect to the RMS of that dimension, estimated from the training sample. The overall scale of the volume is adjusted individually for each test event such that the total number of events confined in the volume lies within a user-defined range (options `NEventsMin/Max`). The adjustment is performed by the class `RootFinder`, which is a C++ implementation of Brent's algorithm (translated from the CERNLIB function RZERO). The maximum initial volume (fraction of the RMS) and the maximum number of iterations for the root finding is set by the options `InitialScale` and `MaxVIterations`, respectively. The requirement to collect a certain number of events in the volume automatically leads to small volume sizes in strongly populated phase space regions, and enlarged volumes in areas where the population is scarce.

Although the adaptive volume adjustment is more flexible and should perform better, it significantly increases the computing time of the PDERS discriminant. If found too slow, one can reduce the number of necessary iterations by choosing a larger `NEventsMin/Max` interval.

### Event weighting with kernel functions

One of the shortcomings of the original PDERS implementation is its sensitivity to the exact location of the sampling volume boundaries: an infinitesimal change in the bound-

ary placement can include or exclude a training event, thus changing $r(i, V)$ by a finite amount.[16] In addition, the shape information within the volume is ignored.

Kernel functions mitigate these problems by weighting each event within the volume as a function of its distance to the test event. The farer it is away, the smaller is its weight. The following kernel functions are implemented in TMVA, and can be selected with the option `KernelEstimator`.

- `Box`
  Corresponds to the original rectangular volume element without application of event weights.

- `Sphere`
  A hyperelliptic volume element is used without application of event weights. The hyperellipsoid corresponds to a sphere of constant fraction in the `MinMax` or `RMS` metrics. The size of the sphere can be chosen adaptive, just as for the rectangular volume.

- `Teepee`
  The simplest linear interpolation that eliminates the discontinuity problem of the box. The training events are given a weight that decreases linearly with their distance from the centre of the volume (the position of the test event). In other words: these events are convolved with the triangle or tent function, becoming a sort of teepee in multidimensions.

- `Gauss`
  The simplest well behaved convolution kernel. The width of the Gaussian (fraction of the volume size) can be set by the option `GaussSigma`.

Other methods implemented for test purposes are "Sinc" and "Lanczos" functions $\propto \sin x/x$ of different (symmetric) orders. They exhibit strong peaks at zero and oscillating tails. The Gaussian and Teepee kernel functions are shown in Fig. 9.

### 5.3.3 Ranking

The present implementation of Likelihood does not provide a ranking of the input variables.

### 5.3.4 Performance

As opposed to many of the more sophisticated data-mining approaches, which tend to present the user with a "black box", PDERS is simple enough that the algorithm can be easily traced and tuned by hand. PDERS can yield competitive performance if the number of input variables is not to large and the statistics of the training sample is ample. In particular, it naturally deals with complex nonlinear variable correlations, the

[16] Such an introduction of artefacts by having sharp boundaries in the sampled space is an example of Gibbs's phenomenon, and is commonly referred to as *ringing* or *aliasing*.

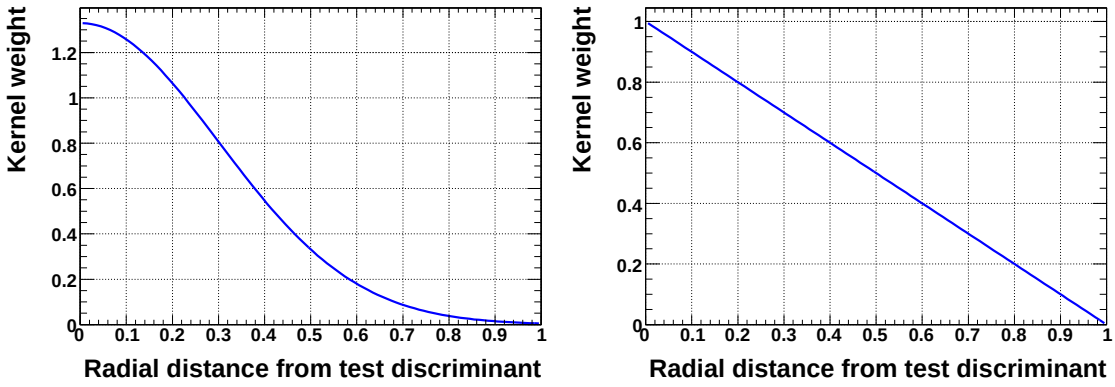


Figure 9: Kernel functions (left: Gaussian, right: Teepee) used to weight the events that are found inside the reference volume of a test event.

reproduction of which may, for example, require involved neural network architectures.

PDERS 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. This is not the case for the other classifiers implemented in TMVA (with some exception for Boosted Decision Trees).

## 5.4 H-Matrix ($\chi^2$) estimator

The origins of the H-Matrix approach dates back to works of Fisher and Mahalanobis in the context of Gaussian classifiers [14, 15]. 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 that exploits differences in the mean values of the vector elements between the two classes for the purpose of discrimination.

The H-Matrix classifier as it is implemented in TMVA is equal or less performing than the Fisher discriminant (see Sec. 5.5), and has been only included for completeness.

### 5.4.1 Booking options

The H-Matrix discriminant is booked via the command:

```
factory->BookMethod( Types::kHMatrix, "HMatrix", "<options>" );
```

Code Example 26: Booking of the H-Matrix classifier: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

No configuration options in addition to those described in Option Table 1 are implemented for the H-Matrix classifier.

### 5.4.2 Description of the classifier and its implementation

For an event $i$, each one $\chi^2$ estimator ($\chi^2_{S(B)}$) is computed for signal ($S$) and background ($B$), using estimates for the sample means ($\overline{x}_{S(B),k}$) and covariance matrices ($C_{S(B)}$) obtained from the training data

$$\chi^2_U(i) = \sum_{k,\ell=1}^{n_{\rm var}} (x_k(i) - \overline{x}_{U,k})\, C^{-1}_{U,k\ell}\, (x_\ell(i) - \overline{x}_{U,\ell}) \ , \tag{16}$$

where $U = S, B$. From this, the discriminant

$$y_H(i) = \frac{\chi^2_B(i) - \chi^2_S(i)}{\chi^2_B(i) + \chi^2_S(i)} \, , \tag{17}$$

is computed to discriminate between the signal and background classes.

### 5.4.3 Ranking

The present implementation of the H-Matrix discriminant does not provide a ranking of the input variables.

### 5.4.4 Performance

The TMVA implementation of the H-Matrix classifier has been shown to underperform in comparison with the corresponding Fisher discriminant (cf. Sec. 5.5), when using similar assumptions and complexity. It is therefore depreciated.

## 5.5 Fisher discriminants (linear discriminant analysis)

In the method of Fisher discriminants [14] event selection is performed 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) upon 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.

### 5.5.1 Booking options

The Fisher discriminant is booked via the command:

```
factory->BookMethod( Types::kFisher, "Fisher", "<options>" );
```

Code Example 27: Booking of the Fisher discriminant: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

The configuration options for the Fisher discriminant are given in Option Table 5.

| Option | Values | Description |
|---|---|---|
| `Method` | `Fisher*, Mahalanobis` | Variations of Fisher discriminants |

Option Table 5: Configuration options for the Fisher discriminant. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The options in Option Table 1 can also be configured.

### 5.5.2 Description of the classifier and its implementation

The classification of the events in signal and background classes relies on the following characteristics: the overall sample means $\overline{x}_k$ for each input variable $k = 1, \ldots, n_{\rm var}$, the class-specific sample means $\overline{x}_{S(B),k}$, and 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)[17].

The *Fisher coefficients*, $F_k$, are then given by

$$F_k = \frac{\sqrt{N_S N_B}}{N_S + N_B} \sum_{\ell=1}^{n_{\rm var}} W_{k\ell}^{-1} \left(\overline{x}_{S,\ell} - \overline{x}_{B,\ell}\right) \ , \tag{18}$$

[17] The within-class matrix is given by

$$W_{k\ell} = \sum_{U=S,B} \langle x_{U,k} - \overline{x}_{U,k} \rangle \langle x_{U,\ell} - \overline{x}_{U,\ell} \rangle = C_{S,k\ell} + C_{B,k\ell} \ ,$$

where $C_{S(B)}$ is the covariance matrix of the signal (background) sample. The between-class matrix is obtained by

$$B_{k\ell} = \frac{1}{2} \sum_{U=S,B} \left(\overline{x}_{U,k} - \overline{x}_k\right) \left(\overline{x}_{U,\ell} - \overline{x}_\ell\right) \ ,$$

where $\overline{x}_{S(B),k}$ is the average of variable $x_k$ for the signal (background) sample, and $\overline{x}_k$ denotes the average for the entire sample.

where $N_{S(B)}$ are the number of signal (background) events in the training sample. The Fisher discriminant $y_{\mathrm{Fi}}(i)$ for event $i$ is given by

$$y_{\mathrm{Fi}}(i) = F_0 + \sum_{k=1}^{n_{\mathrm{var}}} F_k x_k(i)\,. \tag{19}$$

The offset $F_0$ centers the sample mean $\overline{y}_{\mathrm{Fi}}$ of all $N_S + N_B$ events at zero.

Instead of using the within-class matrix, the Mahalanobis variant determines the Fisher coefficients as follows [15]

$$F_k = \frac{\sqrt{N_S N_B}}{N_S + N_B} \sum_{\ell=1}^{n_{\mathrm{var}}} C_{k\ell}^{-1} \left(\overline{x}_{S,\ell} - \overline{x}_{B,\ell}\right)\,, \tag{20}$$

where $C_{k\ell} = W_{k\ell} + B_{k\ell}$.

### 5.5.3 Ranking

The Fisher discriminant analysis aims at simultaneously maximising the between-class separation while minimising the within-class dispersion. A useful measure of the discrimination power of a variable is therefore given by the diagonal quantity $B_{kk}/C_{kk}$, which is used for the ranking of the input variables.

### 5.5.4 Performance

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 (cf. the standard toy example that comes with TMVA).

On the other hand, no discrimination at all 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, if a variable $x \in [-1, 1]$ has a a signal distributions of the form $x^2$, and a uniform background distributions, their mean value is zero in both cases, leading to no separation. The simple transformation $x \to |x|$ renders this variable powerful for the use in a Fisher discriminant.

## 5.6 Artifi cial Neural Networks (non-linear discriminants)

An Artificial Neural Network (ANN) is most generally speaking any simulated collection of interconnected 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 $x_1, \ldots, x_{\mathrm{n_{var}}}$ onto a, in case of a signal-versus-background discrimination

problem, one-dimensional space of output variables $y$. The mapping is non-linear if at least one neuron has a non-linear response to its input.

In TMVA three neural network implementations are available to the user. The first was adapted from a FORTRAN code developed at the Université Blaise Pascal in Clermont-Ferrand,[18] 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. All three neural networks are feed-forward multilayer perceptrons.

### 5.6.1 Booking options

#### The Clermont-Ferrand neural network

The Clermont-Ferrand neural network is booked via the command:

```
factory->BookMethod( Types::kCFMlpANN, "CF_ANN", "<options>" );
```

Code Example 28: Booking of the Clermont-Ferrand neural network: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the options string. See Sec. 3.1.4 for more information on the booking.

The configuration options for the Clermont-Ferrand neural net are given in Option Table 6.

| Option | Values | Description |
|---|---|---|
| `NCycles` | `3000` | Number of training cycles |
| `HiddenLayers` | `"N-1,N-2,..."` | Specification of the network architecture |

Option Table 6: Configuration options for the Clermont-Ferrand neural net. Values given are defaults. See Sec. 5.6.3 for a description of the network architecture configuration. The options in Option Table 1 can also be configured.

#### The ROOT neural network (class TMultiLayerPerceptron)

This neural network interfaces the ROOT class `TMultiLayerPerceptron` and is booked through the Factory via the command line:

[18] The original Clermont-Ferrand neural network has been used for Higgs search analyses in ALEPH, and background fighting in rare $B$-decay searches by the BABAR Collaboration. For the use in TMVA the FORTRAN code has been converted to C++.

```
factory->BookMethod( Types::kTMlpANN, "TMlp_ANN", "<options>" );
```

Code Example 29: Booking of the ROOT neural network: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

The configuration options for the ROOT neural net are given in Option Table 7.

| Option | Values | Description |
|---|---|---|
| `NCycles` | `3000` | Number of training cycles |
| `HiddenLayers` | `"N-1,N-2,..."` | Specification of the network architecture |

Option Table 7: Configuration options for the ROOT neural network. Values given are defaults. See Sec. 5.6.3 for a description of the network architecture configuration. The options in Option Table 1 can also be configured.

### The MLP neural network

The MLP neural network is booked through the Factory via the command line:

```
factory->BookMethod( Types::kMLP, "MLP_ANN", "<options>" );
```

Code Example 30: Booking of the MLP neural network: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the options string. See Sec. 3.1.4 for more information on the booking.

The configuration options for the MLP neural net are given in Option Table 8.

| Option | Values | Description |
|---|---|---|
| `NCycles` | `3000` | Number of training cycles |
| `HiddenLayers` | `"N-1,N-2,..."` | Specification of the network architecture |
| `Normalise` | `True` | Normalised input variables flag |
| `NeuronType` | `sigmoid*, linear, tanh, radial` | Neuron activation function |
| `NeuronInputType` | `sum, sqsum, abssum` | Neuron input norm (synapsis function) |

Option Table 8: Configuration options for the MLP neural network. Values given are defaults. If predefined categories exist, the default category is marked by a '∗'. See Sec. 5.6.3 for a description of the network architecture configuration. The options in Option Table 1 can also be configured.

### 5.6.2 Description of the classifi ers and their implementation

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 *neuron response function* $\rho$.

#### 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 directional connections from one layer to the immediate next one (see Fig. 10). This kind of neural network is termed *multilayer perceptron*; all neural net 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 $n_{\text{var}}$ input variables and 2 output classes the input layer consists of $n_{\text{var}}$ neurons that hold the input values, $x_1, \ldots, x_{n_{\text{var}}}$, and one neuron in the output layer that holds the output variable, the neural net estimator $y_{\text{ANN}}$.[19] Each directional connection between the output of one neuron and the input of another has an associated weight. The value of the output neuron is multiplied with the weight to be used as input value for the next neuron.

#### Neuron response function

The neuron response function $\rho$ maps the neuron input $i_1, \ldots, i_n$ onto the neuron output (Fig. 11). Often it can be separated into a $\mathcal{R}^n \mapsto \mathcal{R}$ *synapsis function* $\kappa$, and a $\mathcal{R} \mapsto \mathcal{R}$ *neuron activation function* $\alpha$, so that $\rho = \alpha \circ \kappa$. The functions $\kappa$ and $\alpha$ can have the

[19] If two output neurons were used in the output layer, one for signal and the other for background, their output values would be $y_{\text{ANN}}$ and $1 - y_{\text{ANN}}$, respectively.

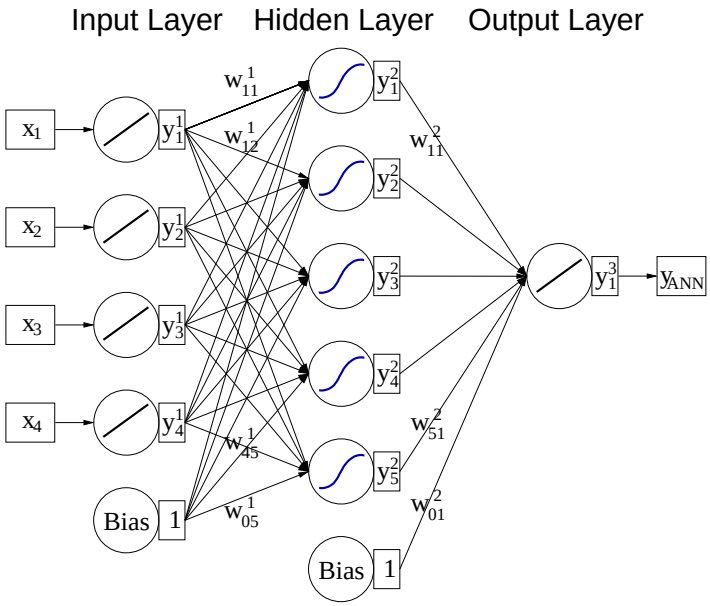


Figure 10: Multilayer perceptron with one hidden layer.

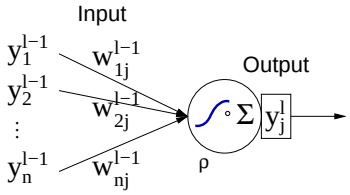


Figure 11: Single neuron $j$ in layer $\ell$ with $n$ input connections. The incoming connections carry a weight of $w_{ij}^{(l-1)}$.

following forms:

$$\kappa: \ (y_1^{(\ell)},..,y_n^{(\ell)}|w_{0j}^{(\ell)},..,w_{nj}^{(\ell)}) \rightarrow \begin{cases} w_{0j}^{(\ell)} + \sum_{i=1}^{n} y_i^{(\ell)} w_{ij}^{(\ell)} & Sum \\ w_{0j}^{(\ell)} + \sum_{i=1}^{n} \left(y_i^{(\ell)} w_{ij}^{(\ell)}\right)^2 & Sum\ of\ squares \\ w_{0j}^{(\ell)} + \sum_{i=1}^{n} |y_i^{(\ell)} w_{ij}^{(\ell)}| & Sum\ of\ absolutes \end{cases} \tag{21}$$

$$\alpha: \ x \rightarrow \begin{cases} x & Linear \\ \frac{1}{1+e^{-kx}} & Sigmoid \\ \frac{e^x - e^{-x}}{e^x + e^{-x}} & Tanh \\ e^{-x^2/2} & Radial \end{cases} \tag{22}$$

### 5.6.3 Network architecture

The number of hidden layers in a network and the number of neurons in these layers are configurable via the option `HiddenLayers`. For example the configuration `"HiddenLayers=N-1,N+10:3"` creates a network with three hidden layers, the first hidden layer with $n_{\text{var}}-1$ neurons, the second with $n_{\text{var}}+10$ neurons, and the third with 3 neurons.

When building a network two rules should be kept in mind. The first is the theorem by Weierstrass ascertaining that for a multilayer perceptron a single hidden layer is sufficient to approximate a given continuous correlation function to any precision, given an arbitrary large number of neurons in the hidden layer. If the available computing power and the size of the training data sample are sufficient, one can thus raise the number of neurons in the hidden layer until the optimal performance is reached.

It is possible that the same performance can be reached with a network with 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.

#### Training of the neural network

The most common algorithm for adjusting the weights that optimise the classification performance of a neural network is the so-called *back propagation.* It belongs to the family of supervised learning methods, where the desired output for every input event is known. Back propagation is used by all neural networks in TMVA. The output of a network (here for simplicity assumed to have a single hidden layer with a Tanh activation function, and a linear activation function in the output layer) is given by

$$y_{\text{ANN}} = \sum_{j=1}^{n_{\text{h}}} y_j^{(2)} w_{j1}^{(2)} = \sum_{j=1}^{n_{\text{h}}} \tanh\left(\sum_{i=1}^{n_{\text{var}}} x_i w_{ij}^{(1)}\right) \cdot w_{j1}^{(2)} \,, \tag{23}$$

where $n_{\text{var}}$ and $n_{\text{h}}$ are the number of neurons in the input layer and in the hidden layer, respectively, $w_{ij}^{(1)}$ is the weight between input-layer neuron $i$ and hidden-layer neuron $j$, and $w_{j1}^{(2)}$ is the weight between the hidden-layer neuron $j$ and the output neuron. Simple summation was used in Eq. (23) as synapsis function $\kappa$.

During the learning process the network is supplied with $N$ training event tuples $\mathbf{x}_a = (x_1, \ldots, x_{n_{\text{var}}})_a$, $a = 1, \ldots, N$. For each training event $a$ the neural network output $y_{\text{ANN},a}$ is computed and compared to the desired output $\hat{y}_a \in \{1, 0\}$ (1 for signal events and 0 for background events). An *error function* $E$, measuring the agreement of the network response with the desired one, is defined by

$$E(\mathbf{x}_1, \ldots, \mathbf{x}_N|\mathbf{w}) = \sum_{a=1}^{N} E_a(\mathbf{x}_a|\mathbf{w}) = \sum_{a=1}^{N} \frac{1}{2} \left(y_{\text{ANN},a} - \hat{y}_a\right)^2 \,, \tag{24}$$

where $\mathbf{w}$ denotes the ensemble of adjustable weights in the network. The set of weights that minimizes the error function can be found using the method of *steepest* or *gradient*

*descent*, provided that the neuron response function is differentiable with respect to the input weights. Starting from a random set of weights $\mathbf{w}^{(\rho)}$ the weights are updated by moving a small distance in $\mathbf{w}$-space into the direction $-\boldsymbol{\nabla}_{\mathbf{w}} E$ where $E$ decreases most rapidly

$$\mathbf{w}^{(\rho+1)} = \mathbf{w}^{(\rho)} - \eta \boldsymbol{\nabla}_{\mathbf{w}} E \,, \tag{25}$$

where the positive number $\eta$ is the *learning rate*.

The weights connected with the output layer are updated by

$$\Delta w_{j1}^{(2)} = -\eta \sum_{a=1}^{N} \frac{\partial E_a}{\partial w_{j1}^{(2)}} = -\eta \sum_{a=1}^{N} (y_{\mathrm{ANN},a} - \hat{y}_a)\, y_{j,a}^{(2)} \,, \tag{26}$$

and the weights connected with the hidden layers are updated by

$$\Delta w_{ij}^{(1)} = -\eta \sum_{a=1}^{N} \frac{\partial E_a}{\partial w_{ij}^{(1)}} = -\eta \sum_{a=1}^{N} (y_{\mathrm{ANN},a} - \hat{y}_a)\, y_{j,a}^{(2)} (1 - y_{j,a}^{(2)}) w_{j1}^{(2)} x_{i,a} \,, \tag{27}$$

where we have used $\tanh' x = \tanh x (1 - \tanh x)$. This method of training the network is denoted *bulk learning*, since the sum of errors of all training events is used to update the weights. An alternative choice is the so-called *online learning*, where the update of the weights occurs at each event. The weight updates are obtained from Eqs. (26) and (27) by removing the event summations. In this case it is important to use a well randomized training sample. Online learning is the learning method implemented in TMVA.

### 5.6.4 Ranking

The MLP neural network implements a variable ranking based on the sum of the weights-squared of the connections that leave the variable input neuron. The importance $I_i$ of the input variable $i$ is given by

$$I_i = \overline{x}_i^2 \sum_{j=1}^{n_{\mathrm{h}}} \left( w_{ij}^{(1)} \right)^2 , \qquad i = 1, \ldots, n_{\mathrm{var}} \,, \tag{28}$$

where $\overline{x}_i$ is the sample mean of input variable $i$.

### 5.6.5 Performance

In the tests we have carried out so far, the MLP and ROOT networks performed equally well, with however 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).

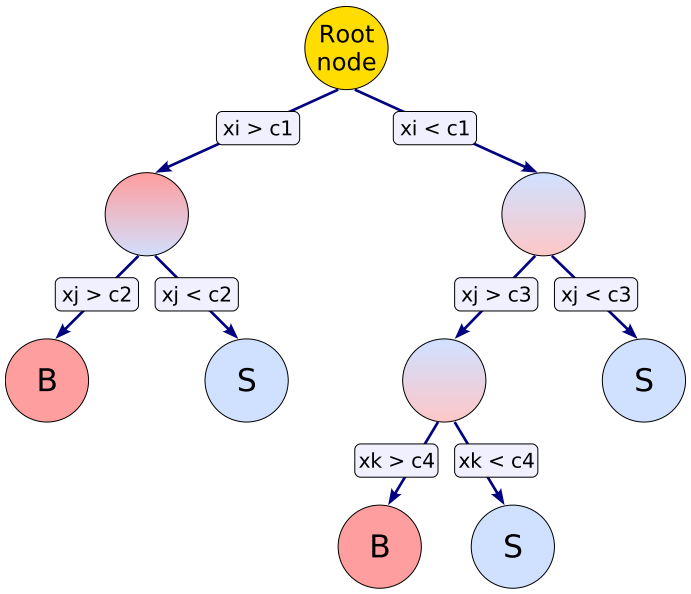


Figure 12: Schematic view of a decision tree. Starting from the root node, a sequence of binary splits using the discriminating variables $x_i$ is performed. Each split uses the variable that at this node gives the best separation between signal and background when being cut on. The same variable may 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.

## 5.7 Boosted Decision Trees

A *decision tree* is a binary tree structured classifier like the one sketched in Fig. 12. Repeated left/right (yes/no) decisions are performed on a single variable at a time until some stop criterion is reached. Like this the phase space is split into regions that are eventually classified as signal or background, depending on the majority of training events that end up in the final *leaf* nodes. The *boosting* of a decision tree (BDT) represents an extension to a single decision tree. Several decision trees (a *forest*), derived from the same training sample by reweighting events, are combined to form a classifier which is given by a (weighted) majority vote of the individual decision trees. Boosting stabilizes the response of the decision trees with respect to fluctuations in the training sample.

### 5.7.1 Booking options

The BDT classifier is booked via the command:

```
factory->BookMethod( Types::kBDT, "BDT", "<options>" );
```

Code Example 31: Booking of the BDT classifier: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

Several configuration options are available to customize the BDT classifier. They are summarized in Option Table 9 and explained in more detail in Sec. 5.7.2.

| Option | Values | Description |
|---|---|---|
| `nTrees` | `200` | Number of trees in the forest |
| `BoostType` | `AdaBoost*, Bagging` | Boosting type for successive tree building |
| `SeparationType` | `GiniIndex*, MisClassificationError, CrossEntropy, SDivSqrtSPlusB` | Separation criterion applied for the node splitting |
| `nEventsMin:` | `10` | Minimum number of events in a node where further splitting is stopped |
| `nCuts:` | `20` | Number of steps in the scan to optimise the cut at a node |
| `UseYesNoLeaf` | `True` | Use simple Yes/No decision from leaf node or (if `False`x) use the training leaf purity as a signal/background weight |
| `UseWeightedTrees` | `True` | Use a weighted (e.g., ln(boost-weight) from AdaBoost) or unweighted majority vote of all trees in the forest |
| `PruneMethod` | `CostComplexity*, ExpectedError` | Pruning method |
| `PruneStrength` | `5` | A parameter to adjust the amount of pruning. It should be large enough such that overtraining is avoided and needs to be tuned for each analysis individually. If it is set to a negative value, an algorithm searches for the optimal prune strength (not necessarily reliable) |

Option Table 9: Configuration options for the BDT classifier. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The common options in Option Table 1 can also be configured.

### 5.7.2 Description of the classifier and its implementation

Decision trees are well known classifiers that allow 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". 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 typically also improves 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 interprete 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.

#### Boosting

Boosting is a general procedure whose application is not limited to decision trees. The same classifier is trained several times using a successively *boosted* (reweighted) training event sample. The final classifier is then derived from the combination of all the individual classifiers. The most popular boosting algorithm is the so-called *AdaBoost* [16] (adaptive boost), where events that were misclassified during the training of a tree are given a higher event weight in the training of the next following tree. Starting with the original event weights when training the first decision tree, the subsequent tree is trained using a modified event sample where the weights of previously misclassified events are multiplied by a common *boost weight* $\alpha$. The boost weight is derived from the misclassification rate err of the previous tree,

$$\alpha = \frac{1 - \mathrm{err}}{\mathrm{err}} \,. \tag{29}$$

The entire event sample is then renormalised to keep the total number of events (sum of weights) in a tree constant.

With the result of an individual tree $h(\mathbf{x})$ ($\mathbf{x}$ being the tuple of input variables) encoded for signal and background as $h(\mathbf{x}) = +1$ and $-1$, respectively, the resulting event classification $y_{\mathrm{BDT}}(\mathbf{x})$ for the boosted classifier is then given by

$$y_{\mathrm{BDT}}(\underline{\mathrm{x}}) = \sum_{i \in \mathrm{forest}} \ln(\alpha_i) \cdot h_i(\mathbf{x}) \,, \tag{30}$$

where the sum is over all trees in the forest. Small (large) values for $y_{\mathrm{BDT}}(\mathbf{x})$ indicate a background-like (signal-like) event. Equation (30) is the default BDT boosting. It can be altered using the option `UseWeightedTrees=False` for which the $y_{\mathrm{BDT}}(\mathbf{x})$ is computed as the average of the individual trees without the weighting factors $\ln(\alpha_i)$.

Another possible modification of Eq. (30) is to use the training purity[20] in the leaf node as respectively signal or background *weights* rather than relying on the binary decision. This option is chosen by setting the option `UseYesNoLeaf=False`. Such an approach however should be adopted with care as the purity in the leaf nodes is sensitive to overtraining and therefore typically overestimated. Tests performed so far with this option did not show significant performance increase. Further studies together with tree pruning are needed to better understand the behaviour of the purity-weighted BDTs.

The other boosting technique implemented in TMVA is a resampling technique, sometimes referred to as *bagging*. It is selected via the `BoostType` option. The resampling is done with replacement, which means that the same event is allowed to be (randomly) picked several times from the parent sample. This is equivalent to regarding the training sample as being a representation of the probability density distribution of the parent event ensemble. If one draws an event out of this ensemble, it is more likely to draw an event from a region of phase-space that has a high cross section, as the original Monte Carlo sample will have more events in that region. If a selected event is kept in the original sample (that is when the same event can be selected several times), the parent sample remains unchanged so that the randomly extracted samples will have the same parent distribution, albeit statistically fluctuated. Training several decision trees with different resampled training data and combining them into a forest results in an averaged classifier that, just as for boosting, is more stable with respect to statistical fluctuations in the training sample. Technically the resampling is implemented by applying random weights to each event of the parent sample.

### Training (Building) a decision tree

The training, building or *growing* of a decision tree is the process that defines the splitting criteria for each node. The training starts with the root node, where an initial splitting criterion for the full training sample is determined. The split results in two subsets of training events that each go through the same algorithm of determining the next splitting iteration. This procedure is repeated until the whole tree is built. At each node, the

[20] The purity of a node is given by the ratio of signal events to all events in that node. Hence pure background nodes have zero purity.

split is determined by finding the variable and corresponding cut value that provides the best separation between signal and background. The node splitting is stopped once it has reached the minimum number of events which is specified in the BDT configuration. The end- or leaf nodes are classified as signal or background according to the class the majority of events belongs to.

A variety of separation criteria can be configured to assess the performance of a variable and a specific cut requirement. Because a cut that selects predominantly background is as valuable as one that selects signal, the criteria are symmetric with respect to the event classes. All separation criteria have a maximum where the samples are fully mixed, i.e., at purity $p = 0.5$, and fall off to zero when the sample consists of one event class only. Tests have revealed no significant performance disparity between the following separation criteria:

- *Gini Index* [default], defined by $p \cdot (1 - p)$.
- *Cross entropy*, defined by $-p \cdot \ln(p) - (1 - p) \cdot \ln(1 - p)$.
- *Misclassification error*, defined by $1 - \max(p, 1 - p)$.
- *Statistical significance*, defined by $S/\sqrt{S + B}$.

The splitting criterion being always a cut on a single variable, the training procedure selects *the* variable and cut value that optimises the *increase* in the separation index between the parent node and the sum of the indices of the two daughter nodes, weighted by their relative fraction of events. The cut values are optimised by scanning over the variable range with a granularity that is set via the option `nCuts`. The default value of `nCuts=20` proved to be a good compromise between computing time and step size. Finer stepping values did not increase noticeably the performance of the BDTs.

In principle, the splitting could continue until each leaf node contains only signal or only background events, which could suggest that perfect discrimination is achievable. However, such a decision tree would be strongly overtrained. To avoid overtraining a decision tree must be *pruned*.

### Pruning a decision tree

Pruning is the process of cutting back a tree from the bottom up after it has been built to its maximum size. Its purpose is to remove statistically insignificant nodes and thus reduce the overtraining of the tree. It has been found to be beneficial to first grow the tree to its maximum size and then cut back, rather than interrupting the node splitting at an earlier stage. This is because apparently insignificant splits can nevertheless lead to good splits further down the tree. TMVA currently implements two tree pruning algorithms.

- For the *expected error pruning* [17] all leaf nodes for which the statistical error estimates of the parent nodes are smaller than the combined statistical error estimates of their daughter nodes are recursively deleted. The statistical error estimate of each

node is calculated using the binomial error $\sqrt{p \cdot (1-p)/N}$, where $N$ is the number of training events in the node and $p$ its purity. The amount of pruning is controlled by multiplying the error estimate by the fudge factor `PruneStrength`.

- *Cost complexity pruning* [18] relates the number of nodes in a subtree below a node to the gain in terms of misclassified training events by the subtree compared the the node itself with *no* further splitting. The cost estimate $R$ chosen for the misclassification of training events is given by the misclassification rate $1 - \max(p, 1-p)$ in a node. The cost complexity for this node is then defined by

$$\rho = \frac{R(\text{node}) - R(\text{subtree below that node})}{\#\text{nodes(subtree below that node)} - 1} \,. \tag{31}$$

  The node with the smallest $\rho$ value in the tree is recursively pruned away as long as $\rho <$ `PruneStrength`.

Note that the pruning is performed *after* the boosting so that the error fraction used by AdaBoost is derived from the unpruned tree.

If the `PruneStrength` option is set to a negative value, an algorithm attempts to automatically detect the optimal strength parameter. The training sample is divided into two subsamples, of which only one is used for training, while the other one serves for validation. Starting with a small value, the prune strength is increased until the maximum performance of the decision tree is reached on the validation sample. This is done for each tree individually. Because of statistical fluctuations the performance may not appear as a smooth function of the prune strength, which could lead to inaccurate optimisation if the validation sample is too small.

### 5.7.3 Ranking

A ranking of the BDT input variables is derived by counting how often the variables are used to split decision tree nodes, and by weighting each split occurrence by the separation gain-squared it has achieved and by the number of events in the node [18]. This measure of the variable importance can be used for a single decision tree as well as for a forest.

### 5.7.4 Performance

Only limited experience has been gained so far with boosted decision trees in HEP. In the literature decision trees are sometimes referred to as the best "out of the box" classifiers. This is because little tuning is required in order to obtain reasonably good results. This is due to the simplicity of the method where each training step (node splitting) involves only a one-dimensional cut optimisation. Decision trees are also insensitive to the inclusion of poorly discriminating input variables. While for artificial neural networks it is typically more difficult to deal with such additional variables, the decision tree training algorithm will basically ignore non discriminating variables as for each node splitting only the best discriminating variable is used. However, the simplicity of decision trees has the drawback

that their theoretically best performance on a given problem is generally inferior to other techniques like neural networks. This is seen for example using the academic training samples included in the TMVA package. For this sample, which has equal RMS but shifted mean values for signal and background and linear correlations between the variables only, the Fischer discriminant provides theoretically optimal discrimination results. While the artificial neural networks are able to reproduce this optimal selection performance the BDTs always fall short in doing so. However, in other academic examples with more complex correlations or real life examples, the BDTs often outperform the other techniques. This is because either there are not enough training events available that would be needed by the other classifiers, or the optimal configuration (i.e. how many hidden layers, which variables) of the neural network has not been specified.

## 5.8 Predictive learning via rule ensembles

This classifier is a TMVA implementation of Friedman-Popscus' RuleFit method described in [19]. 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

$$
\begin{aligned}
r_1(\mathbf{x}) &= I(x_2 < 100.0) \cdot I(x_3 > 35.0)\,, \\
r_2(\mathbf{x}) &= I(0.45 < x_4 < 1.00) \cdot I(x_1 > 150.0)\,, \\
r_3(\mathbf{x}) &= I(x_3 < 11.00)\,,
\end{aligned}
$$

where the $x_i$ are discriminating input variables, and $I(\dots)$ 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 (cf. Sec. 5.7). 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. Hence for the tree illustrated in Fig. 12 a total of 8 rules can be formed. Linear combinations of the rules in the ensemble are created with coefficients (rule weights) calculated using a regularised minimisation procedure [20]. The resulting linear combination of all rules defines a *score* function (see below) which provides the RuleFit response $y_{\mathrm{RF}}(\mathbf{x})$.

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 Fischer discriminant would perform well. To ease the rule optimisation task, a linear combination of the input variables is added to the model. The minimisation procedure will then select the appropriate coefficients for the rules *and* the linear terms. More details are given in Sec. 5.8.2 below.

### 5.8.1 Booking options

The RuleFit classifier is booked via the command:

```
factory->BookMethod( Types::kRuleFit, "RuleFit", "<options>" );
```

Code Example 32: Booking of RuleFit: the first argument is a predefined enumerator, the second argument is a user-defined string identifier, and the third argument is the configuration options string. See Sec. 3.1.4 for more information on the booking.

The RuleFit configuration options are given in Option Table 5.8.1.

### 5.8.2 Description of the classifier and its implementation

As for all TMVA classifiers, the goal of the rule learning is to find a classification function $y_{\mathrm{RF}}(\mathbf{x})$ that optimally classifies an event according to the tuple of input observations (variables) $\mathbf{x}$. The classification function is written as

$$y_{\mathrm{RF}}(\mathbf{x}) = a_0 + \sum_{m=1}^{M_R} a_m f_m(\mathbf{x}) \,, \tag{32}$$

where the set $\{f_m(\mathbf{x})\}_{M_R}$ forms an ensemble of *base learners* with $M_R$ elements. A base learner may be any discriminating function derived from the training data. In our case, they consist of rules and linear terms as described in the introduction. The complete model then reads

$$y_{\mathrm{RF}}(\mathbf{x}) = a_0 + \sum_{m=1}^{M_R} a_m r_m(\mathbf{x}) + \sum_{i=1}^{n_{\mathrm{var}}} b_i x_i \,. \tag{33}$$

To protect against outliers, the variables in the linear terms are modified to

$$x_i' = \min(\delta_i^+, \max(\delta_i^-)) \,, \tag{34}$$

where $\delta_i^{\pm}$ are the lower and upper $\beta$ quantiles of the variable $x_i$. If the variables are used "as is", they may have an unequal *a priori* influence relative to the rules. To counter this effect, the variables are normalised

$$x_i' \to \sigma_r \cdot x_i'/\sigma_i \,, \tag{35}$$

where $\sigma_r$ and $\sigma_i$ are the estimated standard deviations of an ensemble of rules and the variable $x_i'$, respectively.

#### Rule generation

The rules are extracted from a random forest of decision trees. There are several ways to generate a forest. In the current RuleFit implementation it is generated from random subsamples of the training data. Each tree in a forest is generated using a given fraction (`SampleFraction`) of the training sample.[21] If the user gives a fraction $\leq 0$, the fraction

[21] Since both the number of trees and the sample fractions are free parameters, the subsamples used per tree will overlap if the number of trees is greater than 1/`SampleFraction`. By setting `nTrees=-1`, the maximum number of trees allowed without overlapping will be selected. In the output, it is printed whether the subsets are overlapping or not.

| Option | Values | Description |
|---|---|---|
| `Model` | `ModRuleLinear*`, `ModRule`, `ModLinear` | This option controls whether rules and/or linear terms are to be included |
| `LinQuantile` | `0.025` | Concerns only the linear terms and defines a region outside which a variable is regarded as an outlier; a quantile of zero retains all values |
| `MinImp` | `0.01` | Minimum relative importance accepted in the final model |
| `RuleMaxDist` | `0.001` | Minimum "rule distance" allowed; removes similar rules; if zero, all rules are kept |
| `SampleFraction` | `-1` | Event fraction used to train each tree; if $< 0$, the fraction is calculated using Eq. (36) |
| `nTrees` | `-1` | Number of trees in forest; for further description of this and the following three options, see Option Table 9 |
| `nCuts` | `20` | Scan depths of node cut optimisation |
| `SeparationType` | `GiniIndex*`, `MisClassificationError`, `CrossEntropy`, `SDivSqrtSPlusB` | Separation criterion for node splitting |
| `EventsMin` | `0.1` | Minimum fraction giving the minimum number of events in a node where further splitting is stopped |
| `EventsMax` | `0.9` | Ditto, maximum fraction |
| `GDTau` | `0.6` | Minimisation fit threshold; used only if the tau scan range is empty; for the definition of tau, see Sec. 21 |
| `GDTauMin` | `0.0` | Minimum tau in scan |
| `GDTauMax` | `1.0` | Maximum tau in scan |
| `GDNTau` | `1` | Number of tau; if $< 2$, `GDTau` is used |
| `GDTauScan` | `200` | Number of points to scan for best tau along path |
| `GDStep` | `0.01` | Step size along the path |
| `GDNSteps` | `10000` | Maximum number of steps |
| `GDErrScale` | `1.1` | Threshold for error-rate (always $\geq 1$) |

Option Table 10: Configuration options for RuleFit. Values given are defaults. If predefined categories exist, the default category is marked by a '*'. The options in Option Table 1 can also be configured.

is calculated from the training sample size $N$ (signal and background) using the empirical formula [21]

$$f = \min(0.5, (100.0 + 6.0 \cdot \sqrt{N})/N) \,. \tag{36}$$

The topology of each tree is controlled by the configuration parameters `EventsMin`, `EventsMax`, `nCuts` and `SeparationType`, the last two of which are described in Sec. 5.7. The first two parameters define a range of fractions which are used to calculate the minimum number of events in a node required for further splitting. For each tree, a fraction is drawn from a uniform distribution within the given range. The obtained fraction is then multiplied with the number of training events used for the tree, giving the minimum number of events in a node to allow for splitting. In this way both large trees (small fraction) giving complex rules and small trees (large fraction) for simple rules are created. For a given forest of $N_t$ trees, where each tree has $n_\ell$ leaf nodes, the maximum number of possible rules is

$$M_{R,\max} = \sum_{i=1}^{N_t} 2(n_{\ell,i} - 1) \,. \tag{37}$$

To prune similar rules, a *distance* is defined between two *topologically equal* rules. Two rules are topologically equal if their cut sequences follow the same variables only differing in their cut values. The rule distance is then defined by

$$\delta_R^2 = \sum_i \frac{\delta_{i,L}^2 + \delta_{i,U}^2}{\sigma_i^2} \,, \tag{38}$$

where $\delta_{i,L(U)}$ is the difference in lower (upper) limit between the two cuts containing the variable $x_i$, $i = 1, \ldots, n_{\text{var}}$. The difference is normalised to the RMS-squared $\sigma_i^2$ of the variable. Similar rules with a distance smaller than `RuleMinDist` are removed from the rule ensemble. The parameter can be tuned to improve speed and to suppress noise. However, the cut should be used with care since a too large cut value will deplete the rule ensemble and weaken its classification performance.

### Fitting

Once the rules are defined, the coefficients in Eq. (33) are fitted using the training data. For details, the fitting method is described in [20]. A brief description is provided below to motivate the corresponding RuleFit options.

A *loss function* $L(y_{\text{RF}}(\mathbf{x})|\hat{y})$, given by the "squared-error ramp" [20]

$$L(y_{\text{RF}}|\hat{y}) = (\hat{y} - H(y_{\text{RF}}))^2 \,, \tag{39}$$

where $H(y) = \max(-1, \min(y_{\text{RF}}, 1))$, quantifies the "cost" of misclassifying an event of given true class $\hat{y}$. The *risk* $R$ is defined by the expectation value of $L$ with respect to $\mathbf{x}$ and the true class. Since the true distributions are generally not known, the average of $N$ training events is used as an estimate

$$R = \frac{1}{N} \sum_{i=1}^{N} L(y_{\text{RF}}(\mathbf{x}_i)|\hat{y}_i) \,. \tag{40}$$

A line element in the parameter space of the rule weights (given by the vector $\mathbf{a}$ of all coefficients) is then defined by

$$\mathbf{a}(\epsilon + \delta\epsilon) = \mathbf{a}(\epsilon) + \delta\epsilon \cdot \mathbf{g}(\epsilon)\,, \tag{41}$$

where $\delta\epsilon$ is a positive small increment and $\mathbf{g}(\epsilon)$ is the negative derivative of the estimated risk $R$, evaluated at $\mathbf{a}(\epsilon)$. The estimated risk-gradient is evaluated using a sub-sample (Eq. 36) of the training events.

Starting with all weights set to zero, the consecutive application of Eq. (41) creates a path in the $\mathbf{a}$ space. At each step, the procedure selects only the gradients $g_k$ with absolute values greater than a certain fraction ($\tau$) of the largest gradient. The fraction $\tau$ is an *a priori* unknown quantity between 0 and 1. The value $\tau = 0$ implies that at each step on the path all gradients are used, while only the strongest gradient is selected for $\tau = 1$. It is possible to automatically estimate the appropriate $\tau$ by means of a scan. To do so, several paths (`GDNTau`) with different fractions (range `GDTauMin`, `GDTauMax`) are initially scanned for a certain number of points (`GDTauScan`), and the path with the best performance is selected. The step size and the number of points along the paths are given by the options `GDStep` and `GDNSteps`. After every 100 steps, the performance is estimated using the area under the curve of background rejection versus signal efficiency. The area is evaluated using the events *not* used in the calculation of the path (one-fold cross validation). The stepping along the path is stopped when the error rate $\mathrm{e} = 1 - \text{area}$ is larger than ($\texttt{GDErrScale} \cdot \min \mathrm{e}$) and the point giving the minimum error rate is selected. A simple example with a few scan points is illustrated in Fig. 13.

### 5.8.3 Ranking

Since the input variables are normalised, the ranking of variables follows naturally from the coefficients of the model. To each rule $m$ ($m = 1, \ldots, M_R$) can be assigned an importance defined by

$$I_m = |a_m| \sqrt{s_m(1.0 - s_m)}\,, \tag{42}$$

where $s_m$ is the *support* of the rule with the following definition

$$s_m = \frac{1}{N} \sum_{n=1}^{N} r_m(\mathbf{x}_n)\,. \tag{43}$$

The support is thus the average response for a given rule on the data sample. A large support implies that many events pass the cuts of the rule. Hence, such rules cannot have strong discriminating power. On the other hand, rules with small support only accept few events. They may be important for these few events they accept, but they are not in the overall picture. The definition (42) for the rule importance suppresses rules with both large and small support.

For the linear terms, the definition of importance is

$$I_i = |b_i| \cdot \sigma_i\,, \tag{44}$$

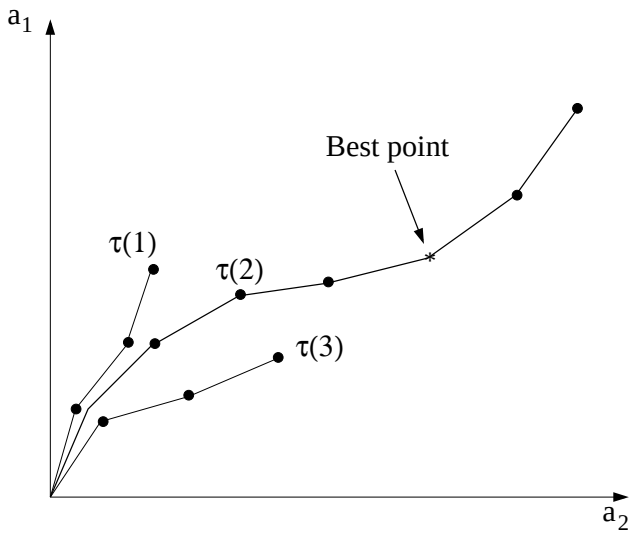


Figure 13: An example of a path scan in two dimensions. Each point represents an $\epsilon$ in Eq. (41) and each step is given by $\delta\epsilon$. The direction along the path at each point is given by the vector $\mathbf{g}$. For the first few points, the paths $\tau(1,2,3)$ are created with different values of $\tau$. After a given number of steps, the best path is chosen and the search is continued. It stops when the best point is found. That is, when the estimated error-rate is minimum.

so that variables with small overall variation will be assigned a small importance.

A measure of the variable importance may then be defined by

$$J_i = I_i + \sum_{m|x_i \in r_m} I_m/q_m \,, \tag{45}$$

where the sum is over all rules containing the variable $x_i$, and $q_m$ is the number of variables used in the rule $r_m$. This is introduced in order to share the importance equally between all variables in rules with more than one variable.

### 5.8.4 Performance

Rule ensemble based learning machines are not yet well known within the HEP community, although they start to receive some attention [22]. Apart from RuleFit [19] other rule ensemble learners exists, such as SLIPPER [23].

The TMVA implementation of RuleFit follows closely the original design described in Ref. [19]. Currently the performance is however slightly less robust than the one of the Friedman-Popescu package. Also, the experience using the method is still scarce at the time of this writing.

To optimise the performance of RuleFit several strategies can be employed. The training consists of two steps, rule generation and rule ensemble fitting. One approach is to modify the complexity of the generated rule ensemble by changing either the number of trees in

the forest, or the complexity of each tree. In general, large tree ensembles with varying trees sizes perform better than short noncomplex ones. The drawback is of course that fitting becomes slow. However, if the fitting performs well, it is likely that a large amount of rules will have small or zero coefficients. These can be removed, thus simplifying the ensemble. The fitting performance can be improved by increasing the number of steps along with using a smaller step size. Again, this will be at the cost of speed performance although only at the training stage. The setting for the parameter $\tau$ may greatly affect the result. Testing with extreme values (close to 0 or 1), will give a feeling for what is a good choice for the given sample. The optimum value will differ with the training sample, so that initially the user is advised to use the automatic scan option to derive the best path.

# 6 Summary and Plans

TMVA is a toolkit that unifies highly customizable multivariate classification algorithms in a single framework thus ensuring convenient use and an objective performance assessment. It is designed for data mining applications in high-energy physics, but not restricted to these. Source code and library of TMVA-v.3.5.0 and higher versions are part of the standard ROOT distribution kit (v5.14 and higher). The newest TMVA development version can be downloaded from Sourceforge.net at http://tmva.sf.net.

This manual introduced the main steps allowing a user to optimise and perform her/his own multivariate analysis. Let us recall the main features of the TMVA design and purpose:

- TMVA works in transparent factory mode to allow an unbiased performance assessment and comparison: all classifiers see the same training and test data, and are evaluated following the same prescription.
- A complete TMVA analysis consists of two steps:
  1. **Training:** the ensemble of available and optimally multivariate customized classifiers are trained and tested on independent signal and background data samples; the classifiers are evaluated and the most performing and concise ones are selected.
  2. **Application:** selected trained classifiers are used for the classification of data samples with unknown signal and background composition.
- A Factory class object created by the user organises the customization and interaction with the classifiers for the training, testing and evaluation phases of the TMVA analysis. The training results together with the configuration of the classifiers are written to result ("weight") files.
- Standardized outputs during the Factory running, and dedicated ROOT macros allow a refined assessment of each classifier's behaviour and performance.

- Once appropriate classifiers have been chosen by the user, they can be applied to data samples with unknown classification. Here, the interaction with the classifiers occurs through a Reader class object created by the user. A classifier is booked by giving the path to its weight file resulting from the training stage. Then, inside the user's event loop, the MVA response is returned by the Reader for each of the booked classifiers, as a function of the event values of the discriminating variables used as input for the classifiers. Alternatively, the user may request from the Reader the probability that a given event belongs to the signal hypothesis.

We give below a summary of the TMVA classifiers, outlining the current state of their implementation, their advantages and shortcomings.

- *Rectangular Cut Optimisation*
  The current implementation is rather advanced. It includes speed-optimised range searches using binary trees, and three optimisation algorithms: Monte Carlo sampling, a Genetic Algorithm and Simulated Annealing. In spite of these tools, optimising the cuts for a large number of discriminating variables remains challenging. The user is advised to reduce the available dimensions to the most significant variables (e.g., using a principal component analysis) prior to optimising the cuts.

- *Likelihood*
  Automatic PDF building through histogram smoothing and approximation with various spline functions and kernel density estimators is implemented.

- *PDERS*
  The multidimensional likelihood approach is in an advanced development stage providing several kernel estimation methods, and speed optimised range search using event sorting in binary trees.

- *Fisher and H-Matrix*
  These are mature algorithms. The Fisher discriminant is linear only in the present implementation. The addition of higher-order moments is considered.

- *Artificial Neural Networks*
  Significant work went into the implementation of fast feed-forward multilayer perceptron algorithms into TMVA. Two external ANNs have been integrated as fully independent methods, and another one has been newly developed for TMVA, with emphasis on flexibility and speed. The performance of the latter ANN (MLP) has been cross checked against the Stuttgart ANN (using as an example $\tau$ identification in ATLAS), and was found to achieve competitive performance.

- *Boosted Decision Trees*
  The BDT implementation has received constant attention over the full year of its development. The current version includes additional features like bagging, and manual or automatic node pruning.

- *RuleFit*
  The original libraries written by J. Friedman are publicly available, but not the source code. We have therefore decided to attempt an independent implementation of this powerful classification approach. The current version achieves almost equivalent results, with however usually somewhat better robustness for the original implementation.

Although TMVA has reached a mature status and has been well tested by many users, there exist limitations that will be worked on for future releases. In particular, the present setup does not allow an unequal number of training events for signal and background. The reason for this restriction is that not all classifiers yet properly handle event weights that deviate from one (though the majority of the classifiers does).

The current emphasis of the TMVA core developments lies on the consolidation and further improvement of the existing classifiers and of the TMVA framework. In spite of that new classifiers are under development. Among these are: a Support Vector Machine, Bayesian classifiers, and a *Committee* classifier, building weighted rules out of arbitrary combinations of TMVA classifiers.

### Acknowledgments

The fast development of TMVA would not have been possible without the contribution and feedback from many developers and users to whom we are indebted. We thank in particular the CERN Summer students Matt Jachowski (Stanford) for the implementation of TMVA's new MLP neural network, and Yair Mahalalel (Tel Aviv) for a significant improvement of PDERS. We are grateful to Doug Applegate, Kregg Arms, René Brun and the ROOT team, Tancredi Carli, Elzbieta Richter-Was, Vincent Tisserand and Marcin Wolter for helpful conversations.

## A Classifier booking

Code Example 33 gives a collection of classifier bookings together with useful default options. These bookings can also be found in the example training job `TMVAnalysis.C`.

```
factory->BookMethod( TMVA::Types::kCuts, "Cuts",
        "MC:EffSel:MC_NRandCuts=100000:AllFSmart" );

factory->BookMethod( TMVA::Types::kLikelihood, "Likelihood",
        "!TransformOutput:Spline=2:NSmooth=5" );

factory->BookMethod( TMVA::Types::kLikelihood, "LikelihoodD",
        "!TransformOutput:Spline=2:NSmooth=5:Preprocess=Decorrelate" );

factory->BookMethod( TMVA::Types::kFisher, "Fisher", "!V:Fisher" );

factory->BookMethod( TMVA::Types::kMLP, "MLP",
        "!V:NCycles=200:HiddenLayers=N+1,N:TestRate=5" );

factory->BookMethod( TMVA::Types::kPDERS, "PDERS",
        "!V:VolumeRangeMode=RMS:KernelEstimator=Teepee:\
         MaxVIterations=50:InitialScale=0.99" );

factory->BookMethod( TMVA::Types::kBDT, "BDT",
        "!V:NTrees=400:BoostType=AdaBoost:SeparationType=GiniIndex:
         nEventsMin=20:nCuts=20:PruneMethod=CostComplexity:
         PruneStrength=3.5:Preprocess=Decorrelate" );

factory->BookMethod( TMVA::Types::kRuleFit, "RuleFit",
        "!V:NTrees=20:SampleFraction=-1:nEventsMin=60:nCuts=20:
         MinImp=0.001:Model=ModLinear:GDTau=0.6:GDStep=0.01:
         GDNSteps=100000:SeparationType=GiniIndex:RuleMaxDist=1e-5" );
```

Code Example 33: Examples for booking classifiers in TMVA. The first argument is a unique type enumerator (the avaliable types can be looked up in `src/Types.h`), the second is a user-defined name (must be unique among all booked classifiers), and the third a configuration option string that is specific to the classifier. For options that are not set in the string default values are used. The syntax of the options should become clear from the above examples. Individual options are separated by a ':'. Boolean variables can be set either explicitly as `MyBoolVar=True/False`, or just via `MyBoolVar/!MyBoolVar`. All concrete option variables are explained in Secs. 4 and 5.

## References


[1] J. Friedman, T. Hastie and R. Tibshirani, "*The Elements of Statistical Learning*", Springer Series in Statistics, 2001.

[2] A. Webb, "*Statistical Pattern Recognition*", 2nd Edition, J. Wiley & Sons Ltd, 2002.

[3] L.I. Kuncheva, "*Combining Pattern Classifiers*", J. Wiley & Sons, 2004.

[4] I. Narsky, "*StatPatternRecognition: A C++ Package for Statistical Analysis of High Energy Physics Data*", physics/0507143 (2005).

[5] The following web pages give information on available statistical tools in HEP and other areas of science: https://plone4.fnal.gov:4430/P0/phystat/, http://astrostatistics.psu.edu/statcodes/.

[6] The BABAR Physics Book, BABAR Collaboration (P.F. Harrison and H. Quinn (editors) *et al.*), SLAC-R-0504 (1998); S. Versillé, PhD Thesis at LPNHE, http://lpnhe-babar.in2p3.fr/theses/these_SophieVersille.ps.gz (1998).

[7] M. Pivk, "*Etude de la violation de CP dans la désintégration* $B^0 \to h^+h^-$ $(h = \pi, K)$ *auprès du détecteur BABAR à SLAC*", PhD thesis (in French), http://tel.archives-ouvertes.fr/documents/archives0/00/00/29/91/index_fr.html (2003).

[8] R. Sedgewick, "*Algorithms in C++*", Addison Wesley, Chapter 26, Boston, USA (1992).

[9] T. Carli and B. Koblitz, Nucl. Instrum. Meth. **A501**, 576-588 (2003); [hep-ex/0211019].

[10] P.J.M. Van Laarhoven and E.H.L. Aart, "*Simulated Annealing: Theory and Application*", D. Reidel Publishing, Dordrecht, Holland, 1987.

[11] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.M. Teller and E. Teller, J. Chem. Phys. **21**, 6, 1087 (1953).

[12] 353QH twice smoothing algorithm, presented by J. Friedman in Proc. of the 1974 CERN School of Computing, Norway, Aug 11-24, 1974.

[13] D.W. Scott, "*Multivariate Density Estimation, Theory, Practice, and Visualization*", Wiley-Interscience, New York, 1992.

[14] R.A. Fisher, Annals Eugenics **7**, 179 (1936).

[15] P.C. Mahalanobis, Proc. Nat. Inst. Sci. India, Part 2A, 49 (1936); P.C. Mahalanobis, "*On the generalized distance in statistics*", Proceedings of the National Institute of Science, Calcutta, **12**, 49 (1936).

[16] Y. Freund and R.E. Schapire, J. of Computer and System Science **55**, 119 (1997).

[17] Y.R. Quinlan, "*Simplifying Decision Trees*", Int. J. Man-Machine Studies, **28**, 221 (1987).

[18] L. Breiman, J. Friedman, R. Olshen and C. Stone, "*Classification and Regression Trees*", Wadsworth (1984).

[19] J. Friedman and B.E. Popescu, "*Predictive Learning via Rule Ensembles*", Technical Report, Statistics Department, Stanford University, 2004.

[20] J. Friedman and B.E. Popescu, "*Gradient Directed Regularization for Linear Regression and Classification*", Technical Report, Statistics Department, Stanford University, 2003.

[21] RuleFit web site: http:/www-stat.stanford.edu/~jhf/r-rulefi t/RuleFit_ help.html.

[22] J. Conrad and F. Tegenfeldt, JHEP **0607**, 040 (2006); [hep-ph/0605106].

[23] W. Cohen and Y. Singer, Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 335, AAAI Press, 1999.

# Index