DRAFT
CMS Internal Note

The content of this note is intended for CMS internal use and distribution only

2009/08/28
Archive Id: 1.5
Archive Date: 2009/05/25 14:41:36

A software framework for Multivariate Analysis Techniques for the CMS offline software

Christophe Saout
CERN

Abstract

This note describes the `MVAComputer` and `MVATrainer` packages found in the CMS offline software (`CMSSW`). These packages provide a modular interface to Multivariate Analysis Techniques (MVA) via a flexible XML-based configuration file. The MVA "computer" packages allows for evaluation of previously trained MVA "networks" from the trainer package. The needed "training expertise" can be stored and retrieved from either plain files for analysis or via the `CMSSW` conditions interface, allowing the data to be store inside the CMS Conditions Database (`CONDDB`) and retrieved via the internet from the central server for run-dependent reconstruction or the online High Level Trigger (HLT). The inner workings of the framework are described and all internally provided algorithms for preprocessing and machine-learning as well as the interface to the third-party package `TMVA` from `ROOT` are described, as well as the various user interfaces.

1 Introduction

The growing field in Multivariate Analysis (MVA) Techniques has seen a huge increase in interest over the last few decades. With the rise of modern computing techniques it has become feasible to employ automated machine-learning technique to replace the tedious procedure of analysing and classifying data by hand.

Multivariate analysis is an umbrella term for techniques and algorithms to statistically analyse data that is typically described in multiple dimensions by a set of variables and is therefore heavily based on the principle of multivariate statistics. MVA techniques can exploit this form of data representation on various levels, like treating each variable individually, exploiting or eleminating inter-variable correlations. Also variables can be treated in a linear fashion or probabilty densities determined on individual variables or on ensembles.

The learning step is performed via algorithms that can be applied to a training sample, which typically includes the truth information, i.e. the quantity the analysis is later on supposed to predict. After this learning step, also called "pattern recognition" (the algorithm is supposed to identify patterns on which its decision-making capabilities are based), the algorithm can then be presented with blind data, i.e. data without the "truth" information, in order to then evalute the data points with respect to the information gained from the training sample. Variations of this concepts also exist, e.g. algorithms that operate on data without truth information, like for instance testing whether newly seen data points agree with previously learned data or simply to find patterns in data. Such techniques are not always considered MVA problems, but can have a certain overlap with the topic.

In a more generalised definition, multivariate analysis techniques reduces, given a model, a large number of variables to a smaller number of variables that are still able to sufficiently describe the problem. It does so by extracting the significant information from the input variables and eventually eliminate correlations, i.e. eliminate redundancy, while doing so.

The typical use case for multivariate analysis is prediction. A very popular problem is classification of data. In particle physics, signal and background discrimination is a very popular problem, where the signal is the physics process of interest and the background consists of other physics processes that one wishes to suppress. In this context, classic cut-based event selection is also a very simple form of multivariate analyis, but with the constraint that typically a person decides how to choose the constraints instead of a machine, but of course, machine-based cut optimisation also exists. Particle identification is another typical use case for such techniques and are used in the offline reconstruction software or even the online trigger.

For classification problems, many algorithms typically do not yield a hard decision, but produce a value that can interpreted as a sort of confidence of the algorithm that the data point falls into a certain category, this output is called a discriminator. For some of the more advanced algorithms this discriminator can even be directly interpreted as a probability following frequentist or Bayesian principles. Those algorithms can sometimes even produce a full probability density for the discriminator giving deep statistical insight into the training sample, analysis method and confidence in the prediction.

In terms of such a discriminator a multivariate analysis for signal/background classification is simply a method to find a function $f(x_i)$ with x_i being the values describing a data point such that $f(x_i)$ evalutes to a value close to A if the data point is signal (A typically being 1) and to B if the data point is background (B typically being 0 or -1). Some algorithms implement this optimisation in form of a global minimisation of a so-called "cost function". By tuning this cost function, the training can be biased to produce an optimal result in the region of in-

terest. For other algorithms similar biasing can typically be achieved by carefully weighting the data points of the training sample. The resemblence of this approach to function fitting and regression analysis is no coincidence, but the necessity to work in a sometimes highly multi-dimensional parameter space complicates things and requires special algorithms Such a discriminator can be simply cut upon, i.e. the working point manually selected (in which case the event selection is just a highly sophisticated form of a cut-based analysis) or the discriminator used in a more advanced way, like using it as an input to yet another MVA or using it as a highly discriminating observable that can be used as input to all kinds of other techniques, some of which can exploit the fact that it can obey statistical principles.

Other use cases for the MVA techniques outside of scientific application are e.g. the prediction of consumer behaviour for a new product, stock market prediction and insurance risk analysis.

In this note, the framework, consisting of the "MVAComputer" and the "MVATrainer", which is implemented in CMS offline software suite "CMSSW" is described. It implements a set of classes and framework modules to implement the full chain of machine learning using a modular architecture in which different "variable processors" can be stacked using an XML-based configuration language. The information gathered during training, referred to as "calibration" in the following (a somewhat misleading term which stems from the fact that it is heavily used within the CMS conditions framework) can then be stored in various ways. It can then be retrieved and used to compute the discriminators, which is referred to as "network evaluation" in the following. The latter part is split into a separate package so that the memory footprint of its use in critical environments is kept as small as possible. The basic functionality of the trainer package is focussed around the signal/background discrimination concept, i.e. the "target" variable is a boolean, although individual variable processors inside a network can be given different targets and weights, so that more complex discrimination problems can be formulated. The variable processors natively support widespread methods such as Fisher's Discriminant and a simple one-dimensional Likelihood Ratio method. In addition a variety of variable preprocessing and general transformations are available, like value-range normalisation and PCA-based variable decorrelation. Furthermore there are many tools for dealing with data representations that contain a non-fixed number of variables. The framework allows variables to be omitted as well as listed more than once. There are modules allowing to sort sets of variables, select subsets or to define simple trainer loops over multiple appearaces of variables inside a data point. Furthermore, some of the variable processor can deal with multiple sets of pdf's that can be combined with a category selector that is based upon a freely configurable set of simple cuts. Including more sophisticated variable processors is achieved through a plugin architecture. By default, a copy of the simple and efficient artificial neural network (ANN) "MLPfit" [**?**] is included in the trainer package as well as an interface to the "TMVA" [**?**] package that is bundled with "ROOT" [**?**], which adds a multitude of more modules, e.g. Boosted Decision Trees (BDT).

2 Overview

The MVA framework consists of two mostly self-contained C++ packages, the "MVAComputer" and the "MVATrainer", with the latter depending on the first. These two packages are found in the "PhysicsTools" subsystem of `CMSSW`. The calibration objects produced by the trainer are represented as C++ objects and their definition is located in the "CondFormats/PhysicsToolsObjects" package.

The two packages contain a plentitude of additional classes and helpers that extend the base functionality and mostly add additional interfaces and framework modules to be easier used

within the surrounding software framework. Besides the plain C++ interface, there are interfaces to ROOT's Cint command line and macro interpreter, interfaces to automatically retrieve or store values from or to ROOT trees, to handle trainer calibration object storage and retrieval via CMSSW's "EventSetup" mechanism or to integrate the trainer with CMSSW's event handling main loop.

The only core dependency of the framework is currently ROOT for the purpose of using its data streamer to serialize and restore a calibration object to a binary large object (BLOB), i.e. to a plain file. Another place where ROOT is used is in some of the variable processors in the trainer for matrix operations.

The modular design of the framework allows the user to define arbitrary "networks" by plugging together "variable processors" to perform specific tasks on a set of variables. Each variable processor has, depending on the type and configuration parameters, a certain number of input variables and output variables. In addition the network as a whole has a set of input and output variables. In fact the whole network always has to have exactly one out variable, namely the network discriminator. This restriction might be lifted in the future to allow the network to output more than a single variable. Inside the network each input variable has to be connected to one of the global input variables or to the output variable of another network and the same rule applies to the global network output variable. These interconnections are configurable. Some restrictions might apply to scheme as not all variable processors always accept all types of variables, more of these details will be discussed later.

The variable processors for the MVA trainer and computer essentially come pair-wise, as for each variable processor used in the computer, a variable processor in the trainer must exist to produce the respective calibration object. Some of those trainer processors do not need any training but are just dummies responsible for passing on the configuration.

In addition to the variables needed for evaluation, the trainer modules expect one or two additional variables, namely the "target" and the "weight". The target is a boolean variable (0 or 1) and indicates whether the data point describe by the current variables belongs to the signal or background sample. The weight variable describes the weight to be given to the data point during training. It can be defined by the user during the training to bias the training towards a more optimal outcome. This can be used to select a preferred working point by adjusting the signal/background ratio or boosting the network to respect some data points stronger than others. Typically a weight of 2 is for instance equivalent to passing the data point twice. If no weight is specified, the weight defaults to 1.

The exact network layout for the training and the variable processor parameters are specified in a trainer description file in the XML format. When training, the trainer will successively construct the calibration object and fill it with the results of the training. During this procedure, a network layout for evaluation by the computer is constructed that is analoguous to the one used in the trainer. Variables used only for training (in particular target and weight) will be automatically removed. Also, whole variable processors that do not affect the outcome of the network, will be automatically purged. This allows the user to add processors purely for training or trainer monitoring without negatively affecting the performance of the network evaluation in the computer. A few processors are only avaiable in the training, namely those used only for monitoring and without any output variables of their own.

Each trainer variable processor can output monitoring data, which can be histograms, matrices, or any other kind of summary information for the training process. This data is not stored in the calibration object, but collected into a separate ROOT file and can be viewed by either a plain

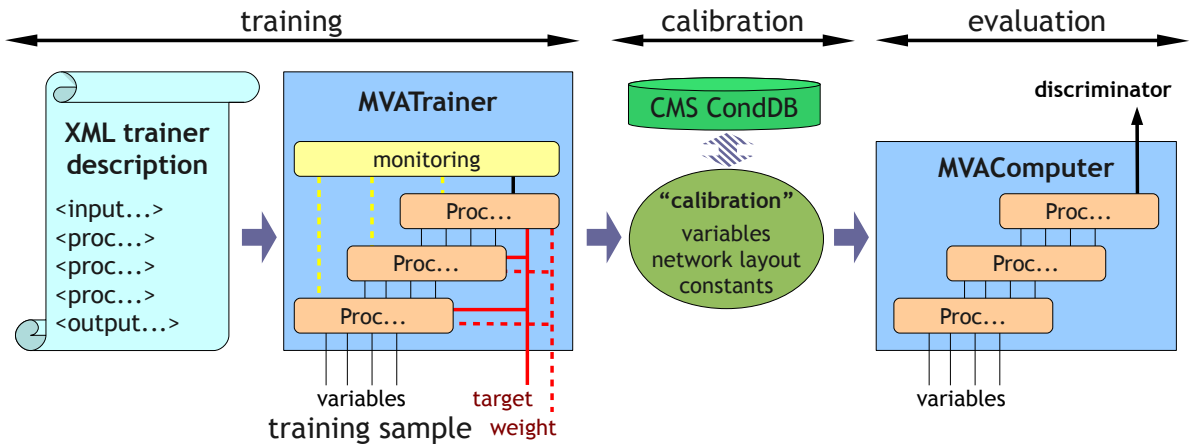

Figure 1: Schematic overview over the MVATrainer / MVAComputer framework

`TBrowser` or the accompanying `ViewMonitoring.C` script or evaluated otherwise.

Figure 1 shows a schematic overview of the framework components.

2.1 Variables

All variables to hold values for the MVA processing are simple floating point variables with double precision. Variable passing to the framework is handled via an unsorted list of identifier-value pairs (the list can be any STL-iterable container), the identifier being essentially a string, i.e. a variable name. For performance reasons the latter is encapsulated into an `AtomicId` object which allow for faster comparison due to internal pointer caching. The variable identifiers are used to uniquely assign the variables to the corresponding input variable slot of the network. If an variable identifier is encountered that is not listed with the network input variables, an error is raised.

An outstanding feature of the framework is the possibility to deal with varying number of variables on a per data point basis. Each variable processor, as well as the whole network, has a fixed number of input variable slots (as defined by the trainer description and stored in the calibration object). Many processors can deal with omission of variables or with variables that appear multiple times (i.e. multiple values with the same identifier). Note that in case of multiple appearances of variables with the same identifier, the relative ordering of the variables with the same identifier becomes relevant, as the values will appear in that order in the variable slots of the processors.

The variable slots of the processors can carry the following capabilities:

- **optional**: variable can be omitted
- **multiple**: variable can appear multiple times

If non of the flags is set, the variable must appear exactly one time, if it is optional, it can appear once or never, if it is multiple, it must appear at least one times, and if both are set it can be omitted or appear one or multiple times.

These flags are propagated automatically within the network, depending on the capabilities of the variable processors. Also, depending on the types of transformations the processors apply to the variables, the output variable slots of the variable processors also carry such flags. Some basic compatibility tests are applied by the network builder during training to ensure no

incompatible variable passing is performed. In addition, the user has to specify the flags for the network input variables manually in the trainer description.

2.2 Variable Processors

The MVA framework provides a number of variables processors, most of which deal with details of variable preparation rather than actual classification. Besides common preprocessing techniques, many of these deal with situations common to event reconstruction problems, like dealing with missing variables, variables that can appear more than once or the need to classify problems into categories beforehand. Also, the modularity allows classifiers to be cascaded or combined in different ways, i.e. to build non-binary classifiers. Complex classification problems can be entirely represented in terms of the MVA framework, reducing the need to prepare the variables from C++ code and, more importantly, allowing the whole evaluation mechanism to be exchanged by simply exchanging the calibration object without code modification.

MVA methods for discriminator computation (classifiers):

- **ProcLikelihood**: Combined Likelihood Ratio using cubic spline representation of the variables' pdf's. Assumes uncorrelated variables and is a $S/(S+B)$ combination of individual variable probabilities.
- **ProcLinear**: A simple linear discriminant from a χ^2 regression fit, also known as Fisher's Discriminant.
- **ProcMLP**: Available via plugin: An Artificial Neural Network (ANN) from the `MLPfit` package, a simple feed-forward network with configurable hidden layers, logistic activation function and several backpropagation learning methods.
- **ProcTMVA**: Access to all methods from `ROOT`'s Toolkit for Multivariate Analysis Techniques (`TMVA`) package.

Variable preprocessing:

- **ProcMatrix**: Linear decorrelation using a matrix rotation defined by Principal Component Analysis (PCA). (lossless)
- **ProcNormalize**: Flattening of value range to $[0,1]$, uses a pdf of all data points to redistribute the values equally in the range. values outside of the range seen in the training dataset are clamped. (lossless with respect to training dataset)
- **ProcLikelihood** (in "individual transformation mode"): Same as for regular **ProcLikelihood**, except that $S/(S+B)$ is computed for each variable individually and not combined into a single likelihood ratio. (lossy if $S/(S+B)$ distribution is not strictly increasing or decreasing)

Tools for dealing with multi-appearance variables:

- **ProcCount**: Counting of the number of variable appearances.
- **ProcForeach**: Looping over a group of variable processors, treating each appearance of a variable from a variable set like an individual data point.
- **ProcOptional**: Replace missing variables with a default value.
- **ProcSort**: Ordering of appearances of variables inside a group of variables by ascending or descending order of the appearances of one of the variables.
- **ProcSplitter**: Splitting of a subset of multi-appearance variables into individual separate variables.

Other:

- **ProcCategory**: Classify a set of variables using a set of rectangular cuts and retur a integer category index.
- **ProcClassed**: Turn an integer category index c into n individual variables: $x_i = 1$ for $i = c$; 0 otherwise.
- **ProcMatrix** (in ranking mode): Perform simple variable correlation analysis and significance ranking based on a linear correlation model.
- **ProcMultiply**: Perform simple variable value multiplication.
- **TreeSaver** (trainer only): Save variables into a `ROOT` tree.

2.3 Calibration object

The calibration object is a plain C++ class instance with `ROOT` dictionaries via `Reflex` to allow serialisation and deserialisation using various interfaces. The two main classes are `MVAComputer` and `MVAComputerContainer` in the `PhysicsTools::Calibration` namespace in the `CondFormats/PhysicsToolsObjects` package.

The first class is a container for exactly one network and contains the list of network input variable, consisting only of the variable identifier, as well as a list of variable processors with configuration and training data, how the variable slots are connected, and which variable to return as the network result. The information contained is kept to a minimum, only the rudimentary information needed to evaluate the network is stored. Calibration objects of this type can be stored and retrieved from plain files or C++ STL streams using the `writeCalibration` and `readCalibration` methods of the `MVAComputer` class.

The `MVAComputerContainer` is a container for multiple networks, where each network is identified by a unique string. Most of the `CMSSW` glue code works with this container object. Individual `MVAComputer` container objects can be stored into and retrieved from this container object.

2.4 Interfaces

The `PhysicsTools/MVAComputer` package provides a number of different interfaces, which are all defined in the `PhysicsTools` namespace. Most of the non-CMSSW related classes are also accessible from `FWLite`

- The core interface: The main class for evaluating a network is `MVAComputer`. The variables passed are of the type `Variable::Value`, with the value list being any STL-iterable container thereof, or the `Varible::ValueList` class for convenience. The variable identifiers are of the type `AtomicId`, which is automatically converted to/from a C or C++ string as needed. The same interface is also used for training a new network, as the MVA trainer framework merely constructs "trainer calibration objects" that will behave like a regular network, but redirect the values to the training processors internally. The big advantage is that the same interface and user code can be used for both training and evaluation, the only difference being the needed target information during training.
- CMSSW EventSetup glue: A predefined `MVAComputerRecord` can be used as EventSetup record for user analyses. Due to the fact that records need to be unique, individial records need to be defined though for reconstruction purposes. Therefore a bunch of helper macros are available to define, register and implement new

records as well as to define EventSetup sources to build `MVAComputerContainer` calibration objects from plain files. Retrieval of calibration objects from the CMS conditions database is supported via the framework's `PoolDBESSource` module.

- The `MVAComputerCache` handles on-demand `MVAComputer` re-instantiation. During a CMSSW process the conditions can in theory change between two events. If this happens, a new `MVAComputer` has to be instantiated with the new object. Because it would be too expensive to instantiate the `MVAComputer` with each event, the `MVAComputerCache` tracks the calibration object for changes and automatically updates the computer if necessary. This is particularly useful when using it in conjunction with the trainer, as the calibration object is supposed to change on each iteration anyhow.
- The `MVAModuleHelper` template covers the basic tasks needed to interface the MVA computer from a framework module (EDFilter, EDProducer, EDAnalyzer) including the communication with the EventSetup mechanism. This also covers details needed to use the same interface also for training.
- The `TreeReader` is an interface to `ROOT` trees. It can automatically construct variable lists from tree branches and pass values to an `MVAComputer` for evaluation. In addition, variables can be added in a `TTree::Fill`-like interface via pointers to variables. This also allows the `TreeReader` interface to be used as a `TTree`-like interface without any `ROOT` tree to be involved at all. This class is also suited to automatically run a trainer on a `ROOT` tree.
- Two command line tools are available that can be directly called from the command line once the CMSSW environment is set up:
  - `mvaConvertTMVAWeights` is used to wrap an already trained `TMVA` classifier into an MVA calibration object using a single *ProcTMVA*. The first argument is a trained `TMVA` classifier text file to be read, the second argument is the file name of the serialized MVA calibration object to be written. Further arguments are the list of variable names in the order they are defined within the trained `TMVA` classifier.
  - `mvaExtractPDFs` is a tool to extract the histograms from trained *ProcNormalize* and *ProcLikelihood* variable processors. The first argument is a serialized MVA calibration object file and the second argument is the name of a `ROOT` file to write with the histograms. This file will then contain histograms named "proc$X$_*type*Y_*repr*" with X being the index of the variable processor (the names of the processors are not saved in the calibration object, so only the index is available, which should correspond to the order as defined in the original trainer description file), *type* being either "norm" for *ProcNormalize* processor or "sig" and "bkg" for signal and background pdf in *ProcLikelihood* processors. The Y is the index of the histogram inside the variable processor data (see section 4 for the layout of the calibration objects) and *repr* is either "histo" or "spline". The first contains the histogram values as they are stored in the object and the latter is a version of the histogram that has contains many more bins on the x-axis and been smoothed using cubic splines, as it is used during evaluation of the network.

The `PhysicsTools/MVATrainer` package consists of a similar set of classes, complementing the MVA computer by all the functionality needed to perform the multivariate analysis and

create the calibration objects. Similarly, most of the classes are accessible from `FWLite` as well.

- The core functionality is provided by the `MVATrainer` class which sole purpose it is to construct a training network from a trainer description file and repeatedly returning transient "training calibration" objects that have to been fed the training dataset using the `MVAComputer` until the full network is built and trained and the final calibration object can be retrieved.
- The CMSSW `MVATrainerLooper` is a framework "EDLooper" that fully wraps the `MVATrainer` interface, turning it into a framework-wide service. The training calibrations are provided via the EventSetup mechanism, so that the training variables can be passed from an EDAnalyzer (or any other framework module) to an `MVAComputer` in the same way as for plain network evaluation. It also instructs the framework to repeatedly loop over the whole dataset until the training is complete.
- Templates for additional CMSSW glue code are provided for instance to created EventSetup record specific saver modules. In conjunction with the `MVATrainerLooper`, the persistent storage of the resulting calibration object is handled by EDAnalyzers that just fire before completion of the CMSSW job. Templates are available to construct such analyzers to store the calibration objects either via the "PoolDBOutputService" (to store results directly using the CondDB mechanism) or to an MVA file via the `MVAComputer::writeCalibration` mechanism.
- A few helper macros to instatiate and register the templates in one line.
- The `TreeTrainer` that glues a `TreeReader` to the `MVATrainer` to train on `ROOT` trees with one single call.
- Three command line tools:
  - The `mvaExtractor` constructs a trainer description template from an already existing calibration object. Since only partial information is available, the general structure only is reconstructed, arbitrary names given to the variable processors and internal variables, and the configuration is left blank. The first argument is an MVA file, the second one is the output XML file.
  - The `mvaTreeComputer` runs over one more multiple `ROOT` trees, computes the discriminator for each entry using a given MVA calibration file and stores the result, together with a copy of the original branches, into a new ROOT file. The MVA calibration and output `ROOT` file are given as the first and second argument. Further arguments dennote the `ROOT` files and/or trees to read. For the exact syntax see the description in section **??**.
  - The `mvaTreeTrainer` reads in an XML trainer description file, one or multiple `ROOT` trees, runs the MVA trainer on the entries, and writes out an MVA calibration file with the results. These are the two first arguments. Further arguments, like for the `mvaTreeComputer`, contain a list of ROOT trees/files.

3 The MVA Computer

The `MVAComputer` class is used to evaluate MVA networks. The two constructors are:

```
namespace PhysicsTools {
   class MVAComputer {
```

```
      public:
         MVAComputer(const Calibration::MVAComputer *calib);
         MVAComputer(Calibration::MVAComputer *calib,
                     bool owned = false);
         ...
```

The second constructor can instruct the `MVAComputer` to own the calibration objects, which means that they will automatically be deleted by the destructed alongside the `MVAComputer`. The calibration objects contain the network layout, list of variables and the trained variable processors.

Evaluation of the network is then done by calling the `eval` method with either a start and end iterator or a container directly (which is just convenience, since it will call the `start()` and `end()` methods:

```
         template<typename Iterator_t>
         double eval(Iterator_t first, Iterator_t last) const;

         template<typename Container_t>
         double eval(const Container_t &values) const;
```

The return value is the network output, i.e. usually the discriminator.

In addition to the `eval` method, an additional `deriv` methods exist as well, which will also perform a network evaluation and return the network output, but in addition all values in the passed variable list will be modified to contain the derivative of the network output with respect to that variable at the given data point. Note that this only works for variables which only undergo continuous transformations from the network input to the network output, otherwise a derivative of zero will be assumed. This evaluation is more CPU-consuming than the `eval` call.

Since variables are passed towards a towards a template method, arbitrary containers that follow the simple forward iterator concept are supported. This includes plain C arrays or any STL containers, as well as the convenience `Variable::ValueList` class. Each object returned must be of the type `Variable::Value`. A constant reference will do for the `eval` method, whereas the `deriv` method expects a writable value object. In principle, the object does not even need to be a `Variable::Value`, it just needs to implement the methods `getName()`, `getValue()`, and `setValue()` for `deriv`.

The value contains a variable identifier (its name) and a floating point value. The `Variable::Value` is used for this purpose, which can be either constructed using the pair, or the respective setter methods:

```
namespace PhysicsTools {
   class Variable {
      public:
         class Value {
            public:
               Value();
               Value(AtomicId name, double value);

```

```
            void setName(AtomicId name);
            void setValue(double value);

            AtomicId getName();
            double getValue();
            ...
```

The `AtomicId` class is a performance wrapper around essentially a string. It will upon construction insert the new identifier into a private cache or use the entry thereof, if the identifier is already inside the cache. This allows comparisons of two `AtomicId` to be essentially replace a simple pointer comparison with the expense of a lookup at construction time. Hence the name "atomic identifier", since pointer comparisons can be performed atomically on CPU's. For performance reasons it is therefore desirable to construct all identifiers outside of the main loop, if performance is considered critical. Otherwise, `AtomicId`'s can be transparently cast from and to C and STL strings, for instance a string constant to be simply used in places where an identifier is expected.

The `Variable::ValueList` helper behaves like a `std::vector<Variable::Value>` and particularly has this convenience method:

```
        void add(AtomicId id, double value);
```

Here is a very simple example of how this can be used:

```
using namespace PhysicsTools;

MVAComputer mva("training.mva");

Variable::ValueList values;
values.add("x", 4.0);
values.add("y", 3.0);
values.add("z", 5.0);

double discr = mva.eval(values);
```

And here is a more performant example used inside a class that tries to avoid `AtomicID` construction and dynamic memory allocation in the evalMVA method:

```
class Test {
   public:
      Test(const Calibration::MVAComputer *calibration);

      double evalMVA(double x, double y, double z) const;

   private:
      enum Variables { X = 0, Y, Z, MAX_VAR };

      Variable::Value variables[MAX_VAR];

```

```
429       MVAComputer mva;
430 };
431
432 Test::Test(const Calibration::MVAComputer *calibration) :
433    mva(calibration)
434 {
435    variables[X].setName("x");
436    variables[Y].setName("y");
437    variables[Z].setName("z");
438 }
439
440 double Test::evalMVA(double x, double y, double z) const
441 {
442    variables[X].setValue(x);
443    variables[Y].setValue(y);
444    variables[Z].setValue(z);
445
446    return mva.eval(variables, variables + MAX_VAR); // start and end
447 }
```

A variation on this kind of class can be used as evaluation abstraction for particular problems and be used for both evaluation and training. The latter only requires the addition of one or two variables (namely target and weight information).

When called, the `eval` or `deriv` method will test all variable identifiers in the passwd value container against the input variable identifiers defined in the calibration object. If an unknown identifier is encountered, an exception is raised, whereas the other way around, it is not necessary that all variables listed in the calibration object are passed by the user, since the variable specification might be optional (the variable flags are not stored in the calibration object and therefore there is no check for correctness).

In addition, the `MVAComputer` contains static methods to read and write a calibration object from or to a file. It will use `ROOT`'s "TBufferFile" mechanism to use the Reflex dictionaries for the calibration object. This means that in order to use the methods, the dictionaries have to be loaded and initialised by calling the `ROOT::Cintex::Cintex::enable()` method from `ROOT`'s `Cintex` library prior to using this functionality.

```
462         static Calibration::MVAComputer *readCalibration(
463                         const char *filename);
464
465         static Calibration::MVAComputer *readCalibration(
466                         std::istream &is);
467
468         static void writeCalibration(
469                         const char *filename,
470                         const Calibration::MVAComputer *calib);
471
472         static void writeCalibration(
473                         std::ostream &os,
474                         const Calibration::MVAComputer *calib);
```

Both the read and write methods support the use of either a filename or an open STL stream. In addition there are two analoguous constructors that allow an `MVAComputer` to be constructed from such a file or stream directly:

```
MVAComputer(const char *filename);
MVAComputer(std::istream &is);
```

3.1 The TreeReader

The `TreeReader` class has two purposes, firstly to provide a `TTree::Fill`-like interface to the MVA computer evaluation and secondly to optionally use that functionality to actually interface a `ROOT` tree. Therefore, the `TreeReader` differentiates between two kinds of variables, depending on whether they can appear multiple times or not. The reason is that the both the argument passing as well as the `TTree` branch types differ. For variables that can appear at most one time, i.e. variables without the "multiple" flag, a single floating point value is sufficient. For the other case, one needs to revert to an array. The interfaces makes use of an STL vector for this case. The idea is that before "filling" data into the `TreeReader`, all branches are defined. Each branch has a name, i.e. an MVA identifier, a type, "single" or "multiple", and its source can either be a `ROOT` tree branch, or a C++ variables referenced by a pointer. Each call to the `fill` method then collects variables for all registered branches and evalutes the network.

There are two constructors:

```
namespace PhysicsTools {
   class TreeReader {
      public:
         TreeReader();
         TreeReader(TTree *tree, bool skipTarget = false,
                    bool skipWeight = false);
         ...
```

The second constructor is just for convenience and has the same effect as calls to `setTree` and `automaticAdd` (see below).

These two template methods, which are implemented for *Double_t*, *Float_t*, *Int_t* and *Bool_t*:

```
template<typename T>
void addSingle(AtomicId name, const T *value, bool opt = false);

template<typename T>
void addMulti(AtomicId name, const std::vector<T> *value);
```

These methods adds an MVA variable that is assigned a given pointer in memory. It's type is automatically detected and has to match one of the four native `ROOT` types the template is implemented for. For multi-appeareance variables, the *value* argument must point to an STL vector of one of the supported data types. The *opt* argument of the `addSingle` method indicates whether the reader should interpret a magic value as "omitted" variable. If this flag is set to true, by default the magic value of the predefined constant `TreeReader::kOptVal` will be recognized as such. This constant is set to -999. One can use the `setOptional` method to modify this behavior, as well as the magic constant for any of the already added branches:

```
        void setOptional(AtomicId name, bool opt,
                         double optVal = kOptVal);
```

Analoguous these two methods add branches, but instruct the read to retrieve them from an actual `ROOT` tree instead of a memory location:

```
      void addBranch(const std::string &expression,
                     AtomicId name = AtomicId(), bool opt = true);
      void addBranch(TBranch *branch,
                     AtomicId name = AtomicId(), bool opt = true);
```

The first method adds a branch by name *expression*. If the MVA identifier *name* is omitted, the branch name is used as identifier. The *opt* argument, again, turns on the magic value regonition for variable omission. The second method takes an explicit pointer to a `TBranch`. Both methods autodetect the branch type. All data types that are supported by `addSingle` and `addMulti` are supported, i.e. plain value branches or `std::vector<...>`'s of one of the four supported types.

```
        void setTree(TTree *tree);
        void automaticAdd(bool skipTarget = false, bool skipWeight = false);
```

The `setTree` method sets the current `ROOT` tree. If this method (or the corresponding constructor) is never used, the reader can be used with memory locations only. The `automaticAdd` method scans through all branches in the current tree and automatically adds all compatible branches as MVA variables. The *skipTarget* and *skipWeight* options instruct it to ignore branches named `__TARGET__` and `__WEIGHT__` during this procedure, as they play a magic role for trainer purposes.

```
        uint64_t loop(const MVAComputer *mva);
        double fill(const MVAComputer *mva);
        Variable::ValueList fill();
```

The `loop` methods evalutes the MVA network passed via the *mva* argument for all entries in the current tree. The discriminators are never returned and this method is only useful in conjunction with the trainer or the "TreeSaver" variable processor. The first `fill` method passes the variables in the current tree entry and/or the current memory locations to the MVA network in the *mva* argument and returns the discriminator. The second `fill` method (without *mva* argument) will simply collect all variables into a value list and return that one instead.

3.2 EventSetup interaction helper macros

In CMSSW, especially for event reconstruction, it is usually not desirable to retrieve the calibration directly from a file within the EDM module, but rather pass it around as a calibration object via the foreseen framework "EventSetup" mechanism. The objects are loaded or created from an "ESSource" or "ESProducer". The calibration objects are organized in records, which are unique C++ classes registered with the framework and assigned a C++ type.

A few macros from `PhysicsTools/MVAComputer/interface/HelperMacros.h` are intended to help with the registration of such a record:

```
MVA_COMPUTER_CONTAINER_RECORD_DEFINE(XYZRcd);
```

will just declare C++ record class, and this line should be put in the header file for the record definition.

There is also a shortcut available:

```
MVA_COMPUTER_CONTAINER_DEFINE(XYZ);
```

This shortcut is also destined for the headers and will simple add `Rcd` to the name of the container. This goes along with a few other macros that will systematically append different strings to *XYZ* to define records and various framework modules.

```
MVA_COMPUTER_CONTAINER_RECORD_IMPLEMENT(XYZRcd);
```

will register the record with the framework, assign `PhysicsTools::Calibration::MVAComputerContainer` as the type of the calibration object and define a framework plugin. If one wishes, the registration of the record and the creation of the plugin can be done separately:

```
EVENTSETUP_RECORD_REG(XYZRcd);
MVA_COMPUTER_CONTAINER_RECORD_PLUGIN(XYZRcd);
```

The first macro should be put in a library (i.e. under the *src* directory) as it needs to be linked against by all users of the record. The second line defines an EDM plugin, so it can be put in a separate package with `<flags EDMPLUGIN="1">` in the BuildFile. This also means one should in princpile never link against the library defining the plugin. It might for instance get necessary to separate the record registration from the plugin definition if users of the record get scattered over multiple packages. For official records used during reconstruction one is advised to not use these macros at all, but to put all the definitions into the official *CondDB* packages **??**.

The following macro automatically creates a file source plugin that acts as an "ESSource" to load calibration objects from MVA files and register them with an EventSetup record:

```
MVA_COMPUTER_CONTAINER_FILE_SOURCE_IMPLEMENT(XYZRcd, XYZFileSource);
```

For a self-contained package, the following line is just shorthand for evaluating `MVA_COMPUTER_CONTAINER_IMPLEMENT` and `MVA_COMPUTER_CONTAINER_FILE_SOURCE_IMPLEMENT` in one go (and append *Rcd* and *FileSource* to the name automatically):

```
MVA_COMPUTER_CONTAINER_IMPLEMENT(XYZ);
```

Again, this only makes sense for a self-contained package and the caveats mentioned above apply.

Also, a few macros are available to create records not for the `MVAComputerContainer` type, but for individual `MVAComputer` calibration objects. These are, however, deprecated to be used with the EventSetup mechanism and therefore many details are unimplemented for this case.

3.3 EventSetup calibration retrieval from an EDM module

Given the correct definition and registration of an EventSetup record, the retrieval of the object looks like this from the `analyze`, `produce` or `filter` module:

```
void XYZAnalyzer::analyze(const edm::Event& iEvent,
                          const edm::EventSetup& iSetup)
{
   using namespace PhysicsTools;

   edm::ESHandle<Calibration::MVAComputerContainer> handle;
   iSetup.get<XYZRcd>().get(handle);

   const Calibration::MVAComputer *calib = &handle->find("label");
   MVAComputer mva(calib);
   ...
```

Note the use of the `find` method of the calibration container to retrieve an individual MVA calibration using its label.

However, this method has the disadvantage of the MVA computer being constructed for each event, which is unnecessarily wasting CPU time. Although the calibration will typically not change between events, one cannot assume it stays constant over the whole job, and should also not be retrieved in any constructor or other method. The solution is a cache mechanism that constructs an MVA computer on demand and caches it between events. The `MVAComputerCache` provides such a mechanism:

```
class XYZAnalyzer : public edm::EDAnalyzer {
      ...
   private:
      ...
      PhysicsTools::MVAComputerCache mva;
...

void XYZAnalyzer::analyze(const edm::Event& iEvent,
                          const edm::EventSetup& iSetup)
{
   using namespace PhysicsTools;

   mva.update<XYZRcd>(iSetup, "label");
   if (!mva)
      return;
...
```

This code is in its functionality identical to the above. Note that added guard here. In conjunction with the trainer it can happen that the last iteration is dummy and no calibration object exists, so it's better to include this safety check. The `MVAComputerCache` has the pointer dereferencing operators defined and will allow to access the cached `MVAComputer`:

```
   discr = mva->eval(list);
```

4 The variable processors and the calibration object

This section describes all variable processors available to the MVA computer and their representation in the calibration object. The following classes are all defined in `PhysicsTools::Calibration` namespace and the `CondFormats/PhysicsToolsObjects/interface/MVAComputer.h` header file.

A stored MVA network (class `MVAComputer`) contains the following information:

- An ordered list of input variables with their identifiers, which is technically an `std::vector<Variable>` and is called *inputSet*. The class `Variable` is just a wrapper around an `string`.
- An ordered list of variable processors. Each variable processor is represented by its own class, which derives from the `VarProcessor` base class. These processors are stored in a polymorphic container which can be accessed through the *getProcessors* getter in form of a `std::vector<VarProcessor*>` array and added one by one using the *addProcessor* setter. The base class `VarProcessor` contains a variable named *inputSet* which contains a bit array (`BitSet`). How this maps to the variables is described below.
- The index of the global network output variable (*output*). How the variables are indexed is described below.

In order to reconstruct all connections between the input and output variable slots of all variable processors defined in the calibration object, an indexing schema is pursued. In order to keep the calibration object lightweight, no variable names except for the global input variables are saved. The indices 0, 1, ... are assigned to the network input variables in the *inputSet*, in that order. Then the output variables of the first variable processor are added to that list, then the output variables of the second, and so on. This way, each variable used in the network can be assigned a unique index. The *output* variable directly specifies such an index to determine the network discriminator.

The `BitSet` in the base class of each variable processor definition is exactly as many bits large as variables have been collected so far, i.e. the global input variables and the output variables of all variables processors before the current processor. A set bit indicates that this variable should be passed as input variable. Hence, the number of non-zero bits in the bit set corresponds to the number of input variables. It also means that the variables are always passed in their order of appearance. This is a rule that is therefore also enforced in the trainer description (automatic reordering is not performed).

The `MVAComputerContainer` class is a container for multiple `MVAComputer` objects. It is essentially a `string` to `MVAComputer` association. The `add` method returns a reference to an empty newly created `MVAComputer` calibration object that can be filled or set using the assignment operator. The `find` method can be used to retrieve an `MVAComputer` using a given string label.

In addition, in both the `MVAComputer` and its container version the `getCacheId` and `changed` methods can be used to quickly compare calibration objects for equality (mainly used to track whether a calibration object changed between EDM events).

The following is a list of all currently supporte variable processors and the description of the corresponding calibration object class and its variable processor evaluation:

- **ProcCategory** computes a "category index" by returning the index assigned to the

boxes in a rectangular grid that is defined over the space spanned by the n input variables.

type	*number*	*optional*	*multiple*
input variables	$n = variableBinLimits.size()$	no	no
output variables	1	no	no

The calibration object is defined as follows:

```
class ProcCategory : public VarProcessor {
    public:
        typedef std::vector<double> BinLimits;

        std::vector<BinLimits>          variableBinLimits;
        std::vector<int>                categoryMapping;
};
```

The `BinLimits` defines k bins per dimension using an array of $k-1$ entries. The first, inner and last bins are defined as going from $(-\infty, limit_1)$, $[limit_i, limit_{i+1})$ for i $= 1, ..., k-1$ and $[limit_{k-1}, \infty)$ respectively. Each input variable has such a bin limit array assigned in *variableBinLimits* (in order of appearance) and spans an n-dimensional array of size $\prod_{i=1}^{n} k_i$. The array entries are stored in *categoryMapping*, and the index of an entry is computed as $index = [\sum_{i=1}^{n-1}(\prod_{j=i+1}^{n} k_j)(bin_i - 1)] + (bin_n - 1)$ (in more than one dimension, with *index* starting at zero). *ProcCatgory* then returns that integer value, the "category", from the category mapping array at the computed index.

The returned category index is mainly intended as input to *ProcNormalize* and *ProcLikelihood*, but can also be used in other ways.

- **ProcClassed** turns an index variable i into a number of boolean output variables, out of which the i-th variable is 1 and all others 0 (or all zero, if i is out of range, i being counted starting at zero).

type	*number*	*optional*	*multiple*
input variables	1	no	no
output variables	*nClasses*	no	no

```
class ProcClassed : public VarProcessor {
    public:
        unsigned int                    nClasses;
};
```

- **ProcCount** counts the number of variables in each of the input variable slots and returns it in that integer number as output variables.

type	*number*	*optional*	*multiple*
input variables	n	yes	yes
output variables	n	no	no

The calibration object is empty.

- **ProcForeach** modifies the control flow by causing the following *nProcs* variable processors to be executed in a loop, one iteration for each variable appearance of *ProcForeach*'s input variables. Each of the input variables of *ProcForeach* itself is copied to the output variables unmodified, but one-by-one. A prerequisite for this to work

is that all input variable slots containt the same amount of variable appearances, otherwise the behaviour is undefined. Each of the variable processors enclosed in the loop must fulfill the following constraint: All output variables that are produced inside the loop and intended to be picked up from outside of the loop, must always appear exactly one time per iteration (i.e. follow "non-optional" and "non-multiple" rules). If these output variables are then accessed from outside (after) the loop, they will then appear with as many instances as iterations where executed, i.e. the output of the iterations are collected into the variable slots. In addition, the first *ProcForeach* output variable, prepended to each copy of all input variables for the current iteration, will be set to the current iteration number, starting with zero.

type	*number*	*optional*	*multiple*
input variables	n	yes	yes
output variables	1: the iteration index	no*	no*
	n: copies of input variables	no*	no*

*: Outside of the loop the number of appearances is identical to the input variables.

```
class ProcForeach : public VarProcessor {
    public:
        unsigned int                    nProcs;
};
```

- **ProcLikelihood** computes a Likelihood Ratio of the basic form $LR = S/(S+B)$, with $S;B = \prod_{i=1}^{n} pdf_{\text{sig;bkg}}(x_i)$. There are, however, many more features and modes of operation. The common functionality is that for each input variable a pdf for signal and background is saved. This variable processor can deal both with missing and multi-appearance variables. For the latter, all appearances of a variable within a variable slot are using the same pair of pdf's. They are stored as histograms with equidistant binning. They can either be evaluated by direct lookup or by using cubic spline interpolation, controlled by the *useSplines* flag in the *SigBkg* class. By default, one pair of pdf's is stored for each of the n input variable slots. However, *ProcLikelihood* supports pdf sets in categories, so the effective number of pdf pairs stored is $n \cdot_{categories}$. The number of categories itself is not stored, but computed on the fly by dividing the number of stored pdf pairs by the number of input variables actually passed. A mismatch results in an error. If the categorized mode is used, the actual category index can be passed passed in one of the input variables. The index of that input variable is determined by the value of *categoryIdx* (counting starts at zero). All other variables are mapped to the pdf pairs in order of appearance. A category variable index of $c = -1$ indicates that the use of categories is disabled, in which case n input variables are passed, otherwise $n+1$. The index of the signal/background pdf pair from the array to be used for the variable at index i and category c is computed $index = c \cdot n + i$.

  In addition to the category index itself, there are a boolean flags encoded in the upper bits of the *categoryIdx* variable:

bits	*variable*
$0 - 19$	category index
20	logarithmic output
21	per-variable likelihood
22	always provide an output value
23	do not ignore empty pdf content

These bit indices are also available in the `ProcLikelihood::Flags` enumeration. Note that the first 20 bits have to be correctly sign expanded (in case the index is -1) when the upper bits are cut off. The *logarithmic output* in bit 20 means that the discriminator value is not computed in the range $[0,1]$ as $LR = S/(S+B)$ but in the range $(-\infty,\infty)$ instead with the definition $LR_{log} = log(S/B)$. Bit 22 disables the feature that in problematic situations *ProcLikelihood* omits the output variable instead of producing a degenerate value. Such conditions can arise eg. instance if $S+B$ in the denominator gets too small for the result to be numerically well defined. In these cases, $S/(S+B)$ or $log(S/B)$ will be set to 0.5 or 0 respectively, with bit 21 set, or $\pm$99999 if B or S are too small in logarithmic output mode. If bit 23 is not set, variables are ignored in the computation of S and B (i.e. not included in the product) if the pdf value of both signal and background for that variable are zero. Otherwise, the result with be either omitted or set to one of the default values, if result omission is disabled. Finally, bit 21 causes *ProcLikelihood* to output n variables, with the LR computed for each variable appearance individually and independently of the other variables.

Finally, there is the *bias* member, which contains a factor that is multiplied with the signal pdf value before evaluation of the Likelihood Ratio. This will cause the ratio to be "biased" towards 1 or 0, hence the name. Since the pdf contents for signal and background are normalized before evaluation, the bias allows to add back in prior information about the relative frequence of occurence of signal with respect to background events. The *bias* is defined as a vector, so that there is an individual value for each category. In case there are no categories, the vector contains one element. If the vector is empty, no bias is used (i.e. a bias of 1).

type	*number*	*optional*	*multiple*
input variables	$n+1$ if *categoryIdx* ≥ 0	yes	yes
	n if *categoryIdx* < 0	yes	yes
output variables	1 if bit 21 is false	yes[1]	no
	n if bit 21 is true	yes[2]	yes[2]

[1]: unless bit 22 is set

[2]: output variable layout identical to input variables

```
class ProcLikelihood : public VarProcessor {
    public:
        class SigBkg {
            public:
                HistogramF              background;
                HistogramF              signal;
                bool                    useSplines;
        };

        enum Flags {
                kCategoryMax    = 19,
                kLogOutput,
                kIndividual,
                kNeverUndefined,
                kKeepEmpty
        };
```

```
        std::vector<SigBkg>             pdfs;
        std::vector<double>             bias;
        int                             categoryIdx;
};
```

- **ProcLinear** computes a linear combination of the input variables. The n coefficients for the n input variables are stored in the *coeffs* array, the additive constant in the *offset* variable.

type	*number*	*optional*	*multiple*
input variables	n	no	no
output variables	m	no	no

```
class ProcLinear : public VarProcessor {
    public:
        std::vector<double>             coeffs;
        double                          offset;
};
```

- **ProcMatrix** computes the m-dimensional vector of values obtained by multiplication of the n-dimensional vector of input variables with a matrix. The $m \cdot n$ matrix elements are stored in the `Matrix` object *matrix* array, which in addition also explicitly contains the two matrix dimensions n (*columns*) and m (*rows*). The *elements* vector of the `Matrix` class first contains all elements of the first row, then the second row, and so on. The matrix is multiplied from the left to the vertical input variable vector.

type	*number*	*optional*	*multiple*
input variables	n	no	no
output variables	m	no	no

```
class Matrix {
    public:
        std::vector<double>             elements;
        unsigned int                    rows;
        unsigned int                    columns;
};

class ProcMatrix : public VarProcessor {
    public:
        Matrix                          coeffs;
};
```

- **ProcMLP** evaluates a simple feed-forward multilayer perceptron neural network. The n input variables act as network input layer, the additional layers and all weights are defined in the calibration object. The m neurons of the last (the output) layer act as output variables for the processor. Each layer of type `Layer` is defined by a vector of neurons and the activation function. `false` selects linear activation (using the result of the linear combination as neuron output), `true` selects logistic activation using the sigmoid function $f(x) = 1/(1 + exp(-x))$. Each neuron of type `Neuron` defines a weight (i.e. coefficient) for each neuron of the previous layer plus one additional weight used as additive constant. Each neuron is evaluated by computing the linear combination from the neuron output of the previous layer

using the given weights and applying activation function produces the new output of the neuron. Typically, sigmoid activation is used for the hidden layers and linear activation for the output layer (in order not to constrain the result to the range $(0,1)$).

type	*number*	*optional*	*multiple*
input variables	n	no	no
output variables	m	no	no

```
class ProcMLP : public VarProcessor {
    public:
        typedef std::pair<double, std::vector<double> > Neuron;
        typedef std::pair<std::vector<Neuron>, bool>    Layer;

        std::vector<Layer>              layers;
};
```

- **ProcMultiply** defines simple products of input variables. Out of n =*in* input variables, indexed from 0 to $n-1$, m products are defined by specifying the indices of the variables to be multiplied in the `Config` vector of integers for each output variable, held in the *out* member. Note that in case a `Config` vector is empty, the result of product is 1.

type	*number*	*optional*	*multiple*
input variables	n	no	no
output variables	m	no	no

```
class ProcMultiply : public VarProcessor {
    public:
        typedef std::vector<unsigned int> Config;

        unsigned int                    in;
        std::vector<Config>             out;
};
```

- **ProcNormalize** provides a value range distribution normalization. For each variable, a pdf is stored, that is used to define a transformation $(x) = \int_{-\infty}^{x} pdf(x)\,dx / \int_{-\infty}^{+\infty} pdf(x)\,dx$, such that the transformed variable is equally distributed in the range $[0,1]$. For the transformation function to be well-behaved (i.e. continuous and differentiable) the pdf histogram entries are interpolated using cubic splines, just as for *ProcLikelihood*. Furthermore, the transformation function is monotonically increasing over the whole range, and strictly increasing on the range where the pdf is non-vanishing. This means that the transformation is invertible on this range and no information is lost due to the transformation. For values outside of the range the pdf is defined, the transformation will clamp the result to 0 or 1 respectively. One pdf histogram is defined for each input variable in the *distr* vector (which accumulates both signal and background). The number of entries n in this vector defines the number of input and out variables. Missing variables and multiple appearances are supported, the variables appearances are not coalesced and the output variables and appearance will exactly match the input variables. In addition, the *categoryIdx* variable allows for pdf's to be defined in categories. If this variable is non-negative, it specifies the index of the input variable specifying the category (this variable will then be excluded from the transformation and not

appear as output variable), and the number of entries in the *distr* vector can be an exact multiple of the number of input variables, so that a pdf is defined for each combination of input variable and category. In that respect the behaviour is identical to *ProcLikelihood* and documented there in detail.

type	*number*	*optional*	*multiple*
input variables	$n+1$ if *categoryIdx* ≥ 0	yes	yes
	n if *categoryIdx* < 0	yes	yes
output variables	n	yes*	yes*

*: output variable layout identical to input variables

```
class ProcNormalize : public VarProcessor {
    public:
        std::vector<HistogramF> distr;
        int                     categoryIdx;
};
```

- **ProcOptional** provides a simple way to replace missing variables by a default value. It takes a vector of n optional input variables, and returns them as n output variables, which omitted values replaced by a respective "neutral position", as defined in the respective *neutralPos* vector of the same size.

type	*number*	*optional*	*multiple*
input variables	n	yes	no
output variables	n	no	no

[1]: output variable layout identical to input variables

```
class ProcOptional : public VarProcessor {
    public:
        std::vector<double>             neutralPos;
};
```

- **ProcSort** is used for reordering of multiple appearances within variables. Groups of n variables with identical number of variable appearances can be sorted in unison, with one of the variables being used as sort criterium, like the sorting of a table on one column. The number of output variables therefore matches the number of input variables, and one additional variable is inserted in the first place. It contains the same number of appearances as the other variables and holds the reordering index, which is the index of the variable appearance, starting at zero, which is inserted as a hidden first column with the other variables and has been sorted using the same criterium and then returned as additional variable. This processor contains two calibration variables: *sortByIndex* contains the index of the input variable that contains the column that is sorted upon (counting starts at zero) and *descending* indicates whether this column should be sorted in ascending or descending order.

type	*number*	*optional*	*multiple*
input variables	n	yes	yes
output variables	$n+1$	yes*	yes*

*: output variable layout identical to input variables

```
class ProcSort : public VarProcessor {
    public:
        unsigned int                    sortByIndex;
        bool                            descending;
};
```

- **ProcSplitter** separates off the first m appearances of the passed input variables and writes them into new individual output variables. The input variables hereby can be appear in arbitrary numbers or missing. If there are not enough appearances to fill all output variables, i.e. if they appear less than m =*nFirst* times, the output variables that cannot be filled will be missing. If there are more than m appearances, the overflow will we filled into a respective overflow variable, which therefore can be missing or appear multiple times. For instance, if $m = 4$ and there are 7 appearances, the first four values will be written to four individual variables and the fifth overflow variable will contain the remaining three appearances. This means that for n input variables, $n \cdot (m + 1)$ output variables will be produced. First the $m + 1$ output variables for the first input variable are produced, then the same for the second input variable and so on.

type	*number*	*optional*	*multiple*
input variables	n	yes	yes
output variables	$n \cdot m$ (first *nFirst* appeareances)	yes	no
	n (overflow)	yes	yes

```
class ProcSplitter : public VarProcessor {
    public:
        unsigned int                    nFirst;
};
```

- **ProcTMVA** is a wrapper container for `TMVA` classifier weight files. This allows to use trained TMVA classifiers from within the MVA computer. The *method* variable specified the type of TMVA classifier and currently the following list is supported: *Cuts*, *SeedDistance*, *Likelihood*, *PDERS*, *HMatrix*, *Fisher*, *CFMlpANN*, *TMlpANN*, *BDT*, *RuleFit*, *SVM*, *MLP*, *BayesClassifier*, *FDA*, *Committee*. The *variables* array of size n contains the variable names used during classifier training for the n variable processor input variables and are used to correctly assign the variables to the `TMVA` interface. The *store* vector contains the gzipped [**?**] version of the classifier weight file and is decompressed and passed to the classifier in memory upon initialisation. The *ProcTMVA* variable processor is built as a separate framework plugin and loaded on demand using the EDM plugin loading mechanism.

type	*number*	*optional*	*multiple*
input variables	n	no	no
output variables	1	no	no

```
class ProcTMVA : public VarProcessor {
    public:
        std::string                     method;
        std::vector<std::string>        variables;
        std::vector<unsigned char>      store;
};
```

5 The MVA Trainer

The `MVATrainer` class, provided in the `PhysicsTools/MVATrainer` package, provides a way of constructing MVA networks from a training description language and handling the passing of training data to so-called "trainer processors", the counterparts to the "variable processors" in the `MVAComputer`, and to emit complete calibration objects. In order to reduce code and interface duplication, the actual data-passing is handled by the `MVAComputer` the same way evaluation of a network is done, with the difference being that the target, and optionally weight information has to be passed with each data point. The way the `MVATrainer` works is to provide temporary pseudo calibration objects (transient objects, from which an `MVAComputer` can be instantiated, but which cannot be persistently saved) which have to be fed the whole training dataset repeatedly until the `MVATrainer` is able to build the final calibration object. The reason this has to be done in multiple iterations is that the `MVATrainer` does not cache the training data in memory, and e.g. pdf's have to be created in first iterations in order be able to define preprocessing transformations before the classifiers themselves can be trained. As with the `MVAComputer`, a set of helper classes and standalone applications provide additional high-level interfaces to simplify and automate its use.

The general usage pattern for `MVATrainer` looks like:

1. instantiate a new `MVATrainer` using a training description file name
2. eventually adjust some global training parameters
3. try to retrieve a training calibration, if it fails, the final calibration object should be ready and retrievable
4. construct an `MVAComputer` from the training calibration object
5. feed the training dataset to the computer
6. repeat steps 3 to 6 until done

Creation of an `MVATrainer` is done through its constructor:

```
namespace PhysicsTools {
   class MVATrainer {
      public:
         MVATrainer(const std::string &fileName, bool useXSLT = false,
                    const char *styleSheet = 0);
         ...
```

The first argument is a regular filesystem path to an XML trainer description file. The second and third argument specify whether XSLT processing of the XML file should be used, any if yes, which XSLT stylesheet should be used (if the third argument is empty, the default `PhysicsTools/MVATrainer/data/MVATrainer.xsl` is loaded). In the future this will allow for further simplification of the trainer description, and is disable by default for now. Upon construction, the description is loaded and basic checks conducted. The network is being built internally and the training facilities initialized.

```
         void setAutoSave(bool autoSave);
         void setCleanup(bool cleanup);

         void loadState();
         void saveState();
```

setAutoSave sets whether `MVATrainer` should automatically save the states of all trainer processors to disk (on by default). If disabled, a call to the *saveState()* can manually dump the state. Most variable processors that hold an internal state can dump that state into small text files. This can be either used for debugging or for loading that state back into an `MVATrainer` session using the *loadState()* method. Since the state of the variable processors is dumped after each training iteration (with *autoSave*), the training can be aborted and resumed later, and the completed training iterations do not have to be repeated. The naming scheme for these trainer state files can be specified in the trainer description and typically contains the label of the training processor for distinction. The *setCleanup* method specifies whether temporary files should be deleted automatically (off by default).

```
void setMonitoring(bool monitoring);
```

This method is used to enable or disable the monitoring (off by default). The monitoring collects information about the training, like histograms of the variables at each stage, processor-specific information and training results. The result is written to a ROOT file whose name is determined by the trainer description file (a pseudo trainer processor label of "monitoring" is used). Its use can cause a slight slowdown of the training process and will cause one additional iteration at the end of the training to evaluate the network performance.

```
void setCrossValidation(double crossValidation);
void setRandomSeed(UInt_t seed);
```

By default, the monitoring and performance evaluation during training is performed on the same dataset as the training itself. Whit this behaviour, the performance numbers from the monitoring cannot be trusted to much, as they are susceptible to overtraining. Therefore, evaluation of the performance on an independant sample is strongly suggested. The `MVATrainer` monitoring can also be run in that mode and split the passed sample internally. To do so, the *crossValidation* parameter has to be set to a value in the range $(0, 1)$ and indicates the fraction of the data points to be used for the actual training. The remaining data points are then used for the monitoring evaluation. The decision whether to use a given data point for training or monitoring is made randomly, i.e. a pseudo random number in the range $[0, 1)$ is chosen. If the number is below the *crossValidation* parameter, the data point is used for training, otherwise for monitoring. The *setRandomSeed* selects the initial random seed for this process. Note that the random number generator is reset on each training iteration, in order to produce exactly the same splitting decision in each iteration, as it is mandatory that the trainer processors are presented with the exact same training dataset in each iteration! When conducting such a cross-validation it is recommended to re-run the trainer multiple times with different random seeds and to study fluctuations in the monitoring results.

```
Calibration::MVAComputer *getTrainCalibration() const;
void doneTraining(Calibration::MVAComputer *trainCalibration) const;
```

The *getTrainCalibration* methods returns a new transient trainer calibration to construct an `MVAComputer` to start a training iteratrion. If a null pointer is returned, the network has been fully trained and the final calibration object can be retrieved using `getCalibration`. After having passed the whole training dataset, a call to *doneTraining*, passing the used training calibration will terminate the current iteration.

```
Calibration::MVAComputer *getCalibration() const;
```

This method retrieves the final calibration object. It will return a null pointer if the `MVATrainer` is not yet fully trained.

```
static const AtomicId kTargetId;
static const AtomicId kWeightId;
```

These are two constant `AtomicId` labels that can be used to pass the target and weight variable to the `MVAComputer::eval` method. The string representation for these two constants are "__TARGET__" and "__WEIGHT__", however it is preferred to use the constants if posible, to avoid unnecessary calls to the `AtomicId` constructor.

A simple example showing how to use the `MVATrainer` class can be found in the `test` directory in `testMVATrainer.cpp`.

5.1 Network construction and training iterations

The process of building the network to be stored in the calibration object during the training is controlled by the `MVATrainer` scheduler. This scheduler is responsible for the determination of which trainer processors are to be fed with data and that the variables are preprocessed by the correct variable processors.

For instance, if a very simple example is considered, like a *ProcLinear* classifer on top of a *ProcNormalize* preprocessor, the `MVATrainer` will schedule between two and four iterations, depending on the configuration. In case *ProcNormalize* trainer processor is using the default configuration, it needs two iterations to be trained. The *ProcLinear* trainer needs one iteration, and enabled monitoring will require an additional iteration, leading to four iterations in total. The `MVATrainer` will feed the `ProcNormalize` trainer processor the input variables for the first two iterations. For the third iteration, an `MVAComputer` network now containing a calibrated `ProcNormalize` is built and fed the data, so that the `ProcLinear` trainer processor can be executed. Similarly, the fourth and last iteration will use both calibrated variable processors in the computer and pass everything to a trainer processor that does computes the final discriminator evaluation for the monitoring. In the end, the calibrations for the to trained processors are saved to a calibration object. This calibration is pruned of all processors and inter-processor variable connections that are needed solely for monitoring purposes (all processors that do not contribute to the final discriminator). Figure 2 deptics a simplified version of the three different network layouts used during training for this example.

The scheduler operates in a greedy mode, trying to train as many processors in parallel as possible. As can be seen in the depiction, the feeding of data from the `MVAComputer` network that is constructed from the training calibrations obtained from the `MVATrainer` to the trainer processors is done by building particular variable processors that function as glue between the two halves. These glue processors are called "interceptors". They forward the training data to the respective training processors in the trainer and are created implicitly. The variable processors themselves can indicate to the scheduler whether they need further iterations over the training dataset. They get signalled from the scheduler whether a new training iteration is about to be performed or they should output their calibration object (in case they are signalling completed training). The global *loadState* and *saveState* method calls to the trainer are forwarded to respective methods in the trainer processors. Simple trainer processors that do not require any actual training (like *ProcOptional*) can immediately return a calibration and consequently never see any data.

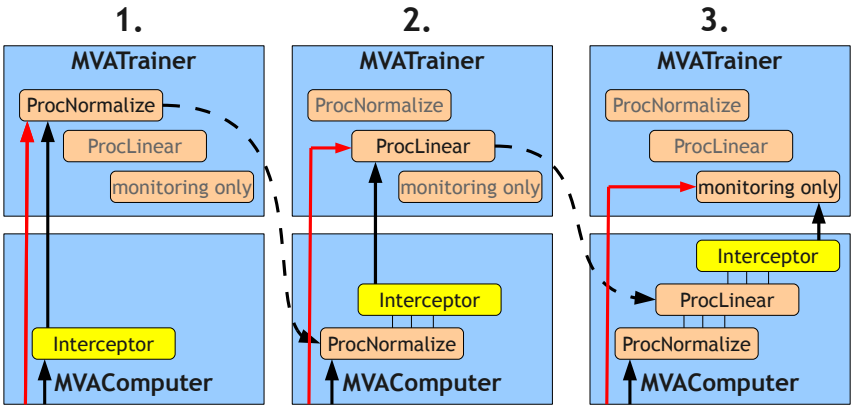

Figure 2: The three different network layouts during training of an example using cascaded *ProcNormalize* and *ProcLinear* processors.

5.2 The TreeTrainer

The `TreeTrainer` class combines the `MVATrainer` with the `TreeReader`, allowing to train a network directory from `ROOT` trees:

```
namespace PhysicsTools {
   class TreeTrainer {
      public:
         TreeTrainer();
         TreeTrainer(TTree *tree, double weight = -1.0);
         TreeTrainer(TTree *signal, TTree *background, double weight = -1.0);
         ...
```

Trees or `TreeReader`'s can be added afterwards using methods described below. Therefore, one is able to construct an empty `TreeTrainer` and the two other constructors are simply convenience. The constructor with only one *tree* argument will expect a tree branch named "__TARGET__" with the target information. The constructor with two separate *signal* and *background* arguments will implictly train with 1 as target for the signal tree and 0 for the background tree. If the optional *weight* parameter is set, it will be used as weight for all data points in the tree. Otherwise, the "__WEIGHT__" branch, if it exists, will be read for weight information.

```
         Calibration::MVAComputer *
         train(const std::string &trainDescription,
               double crossValidation = 0.0,
               bool useXSLT = false);
```

The *train* method will instantiate an `MVATrainer` with with *trainDescription* and optional *crossValidation* and *useXSLT* parameters. Monitoring will be enabled and all other parameters left at their default value. The trees so gar added to the `TreeTrainer` will be read and passed to the trainer until the training is completed and the resulting calibration object returned. This method is convenience as well, if more finegrained control is needed, the following methods should be used instead.

```
void reset();
```

This call resets the `TreeTrainer` back into the same state as a after the default constructor, without any registered trees.

```
void addTree(TTree *tree, int target = -1, double weight = -1.0);
void addReader(const TreeReader &reader);
```

The *addTree* method adds a tree to the list of trees to be used for training. This method is also used by the two convenience constructors. The optional *target* argument passes the value of the target to be used for that tree (0 or 1). If unspecified, a "__TARGET__" branch is expected in the tree. The same applies to the *weight* argument. The *addReader* method allows the user to directly pass a `TreeReader` instead of a tree. This allows the used to configure the reader in detail, like selecting the branches or filling variables from memory.

```
bool iteration(MVATrainer *trainer);
void train(MVATrainer *trainer);
```

The *iteration* method passes the contents of all registered trees to the *trainer* passed as argument. Or more precisely, it requests a training calibration, instantiates an `MVAComputer` and passes all tree entries to its `eval` method. The return value is set to true if the trainer has already been fully trained and no action is performed. The *train* simply repeatedly calls *iteration* until it returns true, i.e. performs a full network training in one go.

5.3 MVA Trainer Executables

The following three command line tools are useful for tasks revolving around MVA training and can be used outside of the framework and without writing any code. They employ standalone MVA calibration object files, ROOT trees and trainer description files as data formats.

- **mvaExtractor** reconstructs a training description skeleton from an existing calibration object. This skeleton includes the list of global input variables, the variable processors and the respective interconnections. It does not contain labels of the variable processors or internal input and output variable names or training processor configurations. The unknown labels are replaced by automatically generated enumerated ones. The command syntax is:

  `mvaExtractor <input.mva> <output.xml>`

  No additional parameters are supported and the input calibration object has to available as a standalone MVA calibration object file.

- **mvaTreeComputer** processes one or more `ROOT` trees through the `MVAComputer` and writes out a new tree, including a copy of all branches and adds a new branch containing the result of an MVA network evaluation. This is particularly powerful to conduct personalized analysis of the discrimination power of the network without having to rely solely on the builtin MVA trainer monitoring, but can also be used for regular network evaluation in environments that work with plain ROOT trees. The command syntax is:

```
Syntax: mvaTreeComputer <input.mva> <output.root> \
                        <input.root> [<input2.root>...]

Trees can be selected as (<tree name>@)<file name>
```

The first argument is an MVA calibration object file, the second argument the name of the output ROOT file. The trees in the output files will be clones of the input trees, so their labelling will be inherited from the input files. The following parameters is a lost of input ROOT files. Specific trees from those files can be selected using the @ syntax *tree@file*, where *tree* is the name of the tree inside the file. This syntax is mandatory if the file contains more than one tree on the top-level directory or the tree is saved in a ROOT file subdirectory. The added branch will be called "__DISCR__", any "__TARGET__" or "__WEIGHT__" branches in the input trees are ignored during evaluation.

- **mvaTreeTrainer** passes one or more ROOT trees through an MVATrainer to train a network.

```
Syntax: mvaTreeTrainer <train.xml> <output.mva> \
                       <data.root> [<data2.root>...]
        mvaTreeTrainer <train.xml> <output.mva> \
                       <signal.root> <background.root>

Recognized parameters:
        -l / --load Load existing training data.
        -s / --no-save Don't save training data.
        -m / --no-monitoring Don't write monitoring plots.
        -w / --no-weights Ignore __WEIGHT__ branches.
        -x / --xslt Use MVATrainer XSLT parsing.
        -v <arg> / --cross-validation <arg>
Use <arg> test/train sample split ratio (0..1).
```

As can be seen, the *mvaTreeTrainer* has a similar calling syntax as the *mvaTreeComputer*. The difference is that an XML trainer description is used as input, and the MVA calibration object file is given as an output file. The fourth and additional arguments specify the input files or input trees (same selection syntax as for *mvaTreeComputer*). Those ROOT trees are expected to contain a "__TARGET__" branch and optionally a "__WEIGHT__" branch with per-entry weights for training. A special case exists if exactly two trees are given on the command line and neither contains a "__TARGET__" branch. In this case the first tree is implictly taken as signal and the second tree as background dataset (i.e. a target of 1 and 0 assumed respectively). The defaults of the TreeTrainer class are used for training. Parameters steer some modifications of the defaults, like disabling the *autoSave* or the monitoring. Additionally, the trainer can be instructed to load the state from trainer processor state dumps (see MVATrainer::loadState() method) before the passing of the training dataset. Also, the trainer can be instructed to ignore eventually found weight branches. The -x parameter can also be used as --xslt=filename.xsl to specify a particular XSLT stylesheet instead of the default one.

5.4 The trainer description language

The trainer language is formulated in XML. The top-level tag is named "MVATrainer". The substructure consists of an optional "general" section with global parameters, a mandatory

"input" section with a list of network input variables, none or more "processor" definitions for the trainer processors and an "output" section to select the network output variable.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<MVATrainer>
   <general>
      ...
   </general>
   <input id="input">
      <var name="x" multiple="false" optional="false"/>
      <var name="y" multiple="false" optional="false"/>
   </input>
   <processor id="..." name="...">
      ...
   </processor>
   <processor id="..." name="...">
      ...
   </processor>
   ...
   <output>
      ...
   </output>
</MVATrainer>
```

- The **general** section consists of a list of global *key* → *value* parameters in the following notation: `<option name="`*option*`">`*value*`</option>`. The current only parameter is the control for the naming of temporary and trainer processor state dump files: "trainfiles". The argument is a `C` printf-style format string with three string arguments:

  1. the trainer processor name
  2. a possible name extension (e.g. in case a variable processors wants to write multiple files)
  3. the file name extension

  This configuration section might look like:

  ```
  <general>
     <option name="trainfiles">train_%s%s.%s</option>
  </general>
  ```

  The `<!--...-->` are XML comments. The three `%s` are the three placeholders fille by the three arguments.
- The global **input** section contains the list of network input variables and their flags. Each entry is a line in the form of `<var name="`*name*`" multiple="`*bool*`" optional="`*bool*`"/>`, where *name* is the variable name as passed to the `MVAComputer` and stored in the calibration object, and the two *bool* are either `true` or `false`, depending on whether the variables may be omitted or appear multiple times. Note that the flags are neither stored in the calibration object nor enforced, so the responsability of following the rules are up to the user. They are only used as hints in certain cases, e.g. for the *TreeSaver* train processor to determine which branch type to use to

save the variable content. The *input* tag itself has to be given an "id" attribute and give it a label. This label is used by the variable processors to select the incoming variables. For consistency it is suggested to always use "input" as label for the global input variables.

- Each train processor is defined in a **processor** section. Each processor is given a label in the "id" attribute which serves the same purpose as in the "input" section, so that the output variables of the processor can be referenced by another variable processor or the output variable selector. Also, each processor definition carries a "name" attribute that contains the name of the trainer processor's `C++` class to load. It currently also corresponds to the name of the variable processor for the `MVAComputer`. This exact correspondance is only for convenience and not a technical restrictions. In the future there might be trainer processors that can output more than one variable processor or build calibrations with variable processors of a different type.

  The order in which the trainer processors are defined is constrained by the availability of the input variables. The variables are only available in order of appearance, i.e. only network input variables and output variables of previously defined trainer processors can be used as input.

  As children of the "processor" tag are, in that order, an "input" tag that selects the input variables, a "config" tag, containing trainer processor specifics and an "output" tag that specifies the output variables. None of the three tags have any further attributes.

  - The "input" tag contains a list of empty "var" tags with both a "source" and a "name" attribute. The "source" tag specifies the label of the section in which the variables was defined. It refers to one of the labels in either the network input section or another variable processor, which are given by the respective "id" attribute. The "name" selects one of the variables define in the respective section. If a variable processor is referred, it selects one of the output variables. A special feature is enabled when a variable with an additional `target="true"` or `weight="true"` is encountered. This variable will not be connected to the variable processor itself, but will cause the variable to be used as target or weight information for that particular trainer processor instead of the user-specified network target or weight. An important detail stems from the fact that a bitmap is used as input variable. This means that the calibration object has no explicit control over the ordering of the variables and instead a convention is used. The convention requires that the variables are always specified in the order of appearance and this corresponds to the order in which they are declared in the XML trainer description. Both the order of the referred sections as well as the order of the variables of the same section have to be respected. If this rule is violated, the `MVATrainer` yields an error message.
  - The "config" tag contents are processor-specific and described in section 4.
  - The "output" section lists the variables emitted by the variable processor. The number of variables cannot be controlled here, it is given by the number of input variables and the configuration section. The role of each variable is defined by the variable processor, see section 4. As a consequence, the only purpose of listing the variables here is to assign them

names that can be referred to later. One variable is specified as `<var name="name"/>`.

A simple example for a variable processor looks like:

```
<processor id="rot" name="ProcMatrix">
   <input>
      <var source="input" name="x"/>
      <var source="norm" name="y"/>
   </input>
   <config>
      <fill signal="true" background="true"/>
   </config>
   <output>
      <var name="rot1"/>
      <var name="rot2"/>
   </output>
</processor>
```

- After the last variable processor, the global network "output" section specifies the network output variable. It consists of a single line `<var source="id" name=name"/>`, like in the input sections of variable processors:

```
<output>
   <var source="linear" name="discriminator"/>
</output>
```

5.5 Trainer Processors

All trainer processors described here that start with "Proc" produce a calibration object for the variable processor of the same name, as described in the section 4. In the following all trainer processors and their configuration are described:

- The **ProcCategory** configuration is directly translated into a calibration object without any training. The category selections done in the configuration are done in a different way than in the calibration object (and then translated to that representation). Here, the user can specify a list of rectangular cuts and the specify the category index that shall be returned if the given data point falls into one such rectangle. In case of overlapping rectangles, the first match is used. In case none of the rectangular selections match, a category index of -1 is returned. The syntax for one group of rectangular selection is:

```
<group>
   <box><range min="min" max="max"/>...</box>
   ...
</group>
...
```

A group consists of an arbitrary number of "boxes", i.e. rectangular selections that result in the same category index. One group needs to be defined for every possible category. The first matching group yields an index of 0, the next one in an index of 1 and so on. Each "box" needs one "range" selection for each of the input variables. A range is given by a minimum and maximum value for each variable, defined in a *min* and *max* attribute. These attributes can be omitted, in which case the range

extends to $-\infty$ and ∞ respectively (in case an empty `<range/>` tag is given, no cut is applied to the variable at all. A box matches if all of the range criteria for all input variables match.

- **ProcClassed** is a training-free processor. The only configuration parameter is defined as `<classes count="`*n*`"/>` with *n* defining the calibration *nClasses* parameter, i.e. the number of boolean output variables.
- **ProcCount** neither needs to be trained or configured, the "config" section is empty.
- **ProcForeach**, as a meta-processor, needs to be configured with the number of following training processors that are to be executed inside the loop. The config line is `<procs next="`*n*`"/>` with *n* being the number of immediately following `<processor...>...</processor>` tags. As far as the calibration object is concerned, the number will be automatically adjusted to the actual amount of variable processors used in the final network (in case it differs from the number of defined training processors).
- The **ProcLikelihood** trainer collects signal and background pdf's for each of the input variables. By default, two iterations are necessary for training. The first iteration determines the lowest and highest value of the variables in the training dataset (the signal and background histograms are assigned the same range, so during range finding no differentiation is made). It is skipped if the user specifies explicit ranges for all variables. The second iteration then fills signal and background histograms.

  ProcLikelihood enables *useSplines* flag in the calibration object by default. So, in order to be able to interpolate between bin centers later on, the histogramming range is slightly increased at both ends, so that the centers of the first and last bin match the minimum/maximum value. As a consequence, the usable bin width of the first and last bin is cut in half. To compensate, the bin contents are doubled.

  Multiple variable appearances for one variable are summed up into the same histograms. The values are filled in with the respective weight from the weight variable. Before storing the histograms into the calibration object, a smoothing can be applied. The smoothing consists of repeated convolution of the histogram contents with a simple smoothing filter with the coefficients $0.1, 0.8, 0.1$, i.e. from each bin 20% of the contents are migrated to the neighboring bins per iteration. For the first and last bin, the migration outside of the histogram is suppressed. The number of smoothing iterations can be specified per variable and defaults to no smoothing.

  The configuration section can contain an optional "general" tag where general parameters can be changed. The parameters themselbes are a list of boolean attributes (i.e. "true" or "false"):

  - Setting *category_bias* to true will cause the trainer to compute a bias for each category individually. (for bias determination see below)
  - Setting *global_bias* to true will cause the trainer to determine the global bias (i.e. summed over all categories).
  - Setting *log* to true will set the *kLogOutput* bit in the calibration object, so that the variable processor will compute the $log(S/B)$ instead of $S/(S+B)$.
  - Setting *individual* to true will set the respective *kIndividual* bit in the calibration object, meaning that a per-variable instead of a combined likelihood ratio is computed.
  - Setting *strict* to true will cause the *kNeverUndefined* bit to be disabled. This

will cause the variable processor to not emit an output variable if a degenerate condition is encountered (see description in section 4).
- Setting *ignore_empty* to false will enable the *kKeepEmpty* bit, in which case the variable processor will not emit an output variable if either signal or background pdf is vanishing.

In addition, the user can manually set a particular global by setting the "bias" attribute in the "general" tag to a particular value.

The next optional tag is the "bias_table" (without attributes). It can be used to manually specify bias values for the individual categories. For each category, one bias in the form of `<bias>`*value*`</bias>` has to be specified. This table, as well as the *category_bias* flag only make sense if categories are used. All biases specified or determined automatically, i.e. the *global_bias*, *category_bias* and *bias* attributes of the "general" tag, as well as the "bias_table" are combined multiplicatively. The determination of the category bias explicitly excludes the global bias, so that both flags have to be enabled together if the absolute signal/background ratios of the categories are to be used in $S/(S+B)$.

After these two optional tags, one tag for each input variable has to be present, in the order they have been listed in the trainer processor "input" section. Regular variables are given an empty "sigbkg" tag, optionally with a few attributes:

- The *size* attribute is given an integer argument and configures the number of bins to be used for histogramming. The default value is 50.
- The *smooth* attribute gives the number of smoothing iterations. The default is not to perform any smoothing.
- The *lower* and *upper* attributes specifiy the minimum and maximum value and override the autodetection. Specifying only one of the two attributes alone is invalid.

In case categories are to be used, the category variable has to be specified with a "category" tag instead. It has to be given the number of categories in its *count* attribute. Specifying more than one "category" variable is invalid.

All histograms produced during training are saved into the monitoring file.

It is suggested to preprocess the variables at least with `ProcNormalize`. The reason is that many distributions are very uneven and with equidistant binning often too much precision is lost due to relevant information being distributed over only few bins and outliers unnecessarily stretching the ranges of the histogram. For a likelihood ratio sometimes it might also be useful to eliminate variable correlations beforehand, e.g. using `ProcMatrix`.

- **ProcLinear** needs one iteration to fill a covariance matrix for all variables and the target information (weights are used for filling). No configuration is required. The matrix coefficients are used to solve the linear regression equation, i.e. that the overall χ^2 with respect to the target is minimized. This linear coefficients produced satisfy the "Fisher's Discriminant" condition.
- **ProcMatrix** also needs one iteration to fill the covariance matrix for all variables. The results can be evaluated in two possible ways. The default is to compute a rotation matrix that diagonalizes the covariance matrix using a Principal Component Analysis (PCA). After this transformation all first order (linear) correlations between any pair of variables are eliminated. The second use case is provide a simple variable importance ranking. In that mode, no output variable is emitted and the processor is

not saved into the calibration object, i.e. in this mode the trainer processor is purely for monitoring.

The configuration constists of a single "fill" tag that can carry a number of boolean attributes:

- Setting *signal* to true causes signal events to be filled into the covariance matrix.
- Setting *background* to true causes background events to be filled into the covariance matrix. Note that if neither *signal* nor *background* are set, both are filled implicitly into the covariance matrix.
- Setting *ranking* will switch *ProcMatrix* to the variable ranking mode. This implicitly enables filling of both signal and background.

The correlation matrix (where each variable-variable correlation is defined in the range $[-1, 1]$) is stored as two-dimensional histogram in the monitoring file.

The ranking mode additional computes the correlation to the target variable and emits that into the monitoring file. In addition it will perform a simple variable ranking defined as follows:

- A simple linear discriminant (Fisher's discriminant) is determined. The advantage is that this can be done by using the already obtained covariance matrix. The correlation of the discriminator to the target is computed, again using already existing matrix components. The value is taken as discriminative power for the whole variable ensemble (as value between 0 and 100 percent).
- The variable that has the smallest contribution to this value is determined. This is achieved by removing one variable at a time from the ensemble and recomputing the ensemble discriminative power using the linear discriminant approach.
- The variable that has been determined to be the least effective is removed from the ensemble. The discriminative power of the ensemble should have reduced by a certain amount. The variable and the loss in discriminative power is documented in the monitoring file.
- The whole procedure is repeated until only one variable is left. This variable is considered the "highest-scoring" variable.

However, one should keep in mind that this procedure is only one possible approach. The advantages are:

- Variable correlations are taken into account. If the ensemble contains two identical variables, the removal of one of the variables does not change the discriminator power of the ensemble. This fact will be recognized by the procedure and suggest to the user to drop one of the variables.
- The ordering of the variables is a good hint at which variables are important and which are not. The loss in ensemble discriminative power for each variable documents how much improvement can be expected when adding in this variable.
- Due to the fact that the computation only relies on one covariance matrix, no looping over data is needed for the algorithm iterations.

However, there are many downsides as well:

- The decision which variable to remove is somewhat arbitrary. It depends

on which variables have already been removed and the decision might have turned out completely differently if another variable had been taken out, since the variable correlations might be very complex. It is therefore strongly suggested to only take the results as a hint and to also have a look at the variable-target correlation of individual variables in the correlation matrix and test out a few other promising combinations by hand. Especially evaluation of variables "by eye" may yield additional characteristics that might make certain variables more suitable than others.

- Only linear correlations are taken into account, and also the linear discriminant cannot take non-linear correlations into account. As a consequence, some portion of discriminative power can be missed. Therefore it is strongly suggested to preprocess the input variables with `ProcNormalize` and `ProcLikelihood` in "individual" mode. This will get the maximum information out of individual variable-target correlations. The inter-variable correlations are still only taken into account linearly.
- Variables can exhibit negative correlations. This can lead suprising effects like the discriminative power of a network slightly growing after removal of a variable. This is an artifact of the linear approach and not a bug. One should try to avoid using variables which do not contribute much, but exhibit unusual correlations.

- **ProcMultiply** is a training-less processor that passes the configuration to the calibration object. The configuration sections contain a list of `<product>`$n * m * ...$`</product>` tags with n, m, ... being input variable indices (starting at zero) and separated by asterisks. One output variable will be emitted per defined product.
- The **ProcNormalize** trainer works in many ways like the *ProcLikelihood* trainer. Histograms with pdf's are filled. The main differences are that no separate histograms for signal and backgrounds are produced and no general flags or bias tables are used. The list of variables is given through a list of "pdf" tags (instead of "sigbkg" tags). The attributes "lower", "upper", "size" and "smooth" have the same meaning as for *ProcLikelihood*. By default, 100 histogram bins and 40 smoothing iterations are used if not overridden. The amount of smoothing is used to reduce the effect of statistical fluctuations and to make the transformation function as smooth as possible. For distributions that have very sharp peaks it might therefore be helpful to reduce the number of iterations. If uncertain, it is best to cross-check the results by looking at the distributions produced as output of the normalization process.

  In addition to the existing attribute, one can additionally specify `signal="true"` or `background="true"` which instructs the trainer processor to only collect signal or background data points into that particular variable. Specifying neither implicitly means that both kind should be filled in (default). In addition to the "pdf" tag, the "category" tag has the same meaning as for *ProcLikelihood*.
- **ProcOptional** is configured by specifying a `neutral` tag for each input variable. Each tag has a *pos* attribute with a floating point value, the "neutral position" as argument. These values are imported into the calibration object directly and no training is required.
- **ProcSort** does not need any training either. The only configuration option is the "key" tag with an integer "index" attribute specifying the index (starting at zero) of the "leader" variable, i.e. the variable which is used as sorting criterium. Optionally,

one can set the "descending" attribute to either "true" of "false", which the latter being the default.

- **ProcSplitter**, another training-less processor, is configured with `<select first="n"/>`, with n being the number of variable appearances to be separated into individual output variables.
- **TreeSaver** is a trainer processor that has no corresponding variable processor. Its sole purpose is to dump variables into a `ROOT` tree. It be used to save either the global input variables directly and used as a simple tool to e.g. extract variables from a framework module (which feeds variables to the `MVAComputer`) into a standalone tree for debugging. Furthermore, writing out variables after being processed by variables processors is also possible.

  The *TreeSaver* therefore supports no output variables. In case the *TreeSaver* is the only module in the trainer description file and no actual MVA networks shall be trained, one has to select a random input variable as network output variable to satisfy the requirements. The *TreeSaver* can, however, be used alongside a regular training, for instance for debugging or in order to conduct some private monitoring ouside of the trainer. Another use case is to write out some preprocessed variables to a tree in order to be able to use the `TreeTrainer` on the resulting tree, avoiding having to redo the training of certain variable processors every time. In conjunction with the state saving and loading feature, one can later one combine all calibration objects into a new network without having to retrain.
- The *ProcMLP* trainer is a `C++` wrapper around the `MLPfit` package [**?**]. It runs in two iterations. The first iteration counts the number of signal and background events and allocates arrays in memory. The second iteration fills these arrays and then calls the neural network trainer. Weights are respected during training. The cost function is a simple χ^2. The average χ^2 per data point is printed on every tenth iteration during training to be able to observe the trend.

  The processor is configured via a `<config steps="n">`*layout*`</config>` tag. n specifies the number of training iterations and *layout* the layout of the hidden layers. The *layout* of the hidden layers is specified by listing the number of neurons per hidden layer, separated by a comma. One hidden layer with about twice the number of neurons as input variables is typically a good starting point. There is an additional *limiter* attribute that can be set to a floating point number. This allows to reduce the amount of data points used for training. In some cases very high statistics are available, which can e.g. be used to get very smooth individual pdf's. Training a neural network however, can take a very long time, so reducing the amount of data while keeping a sensible amount of statistics (twenty to thirty data points per weight, i.e. neuron-neuron connection) might be of interest. Especially if a large background-single discrepancy, that is compensated by weights, is used, one might want to only use every n-th background event. The *limiter* specified the minimum weight that should be passed on to the network trainer. If a data point with a smaller weight is encoutered, a random number is used to unweight the event accordingly, i.e. the ratio between weight and *limiter* is used as the probability to accept the data point (and if it is accepted, its weight is changed to *limiter*).

  The `MLPfit` trainer has a set of tunables, which are currently hardcoded in the *ProcMLP* glue code to sensible defaults (for the exact meaning, see the `MLPfit` package documentation):

parameter	*value*
method	7 (hybrid linear-BFGS backpropagation)
n_{reset}	50
τ	1.5
decay	1.0
η	0.1
Λ	1.0
δ	0.0
ϵ	0.2

- **ProcTMVA** is interfaced by some simple glue code to the trainer. In the first iteration, a `ROOT` tree is written, which is then passed to the configured classifier and the resulting weights file zipped and encapsulated in the calibration object. The branch names for the tree are created by concatenating the variable *source* and *name*, separated by an underscore. The temporary `ROOT` file is removed afterwards if the cleanup option in the trainer is enabled.

  The interface can train multiple classifiers in parallel, however, only the first classifier is actually store in the calibration object. The reason for this is to allow the use of `TMVA`'s builtin comparison tools, even though it is driven by the MVA framework.

  For each classifier to be trained, a "method" tag has to be put into the configuration. The content of this tag is a string with a `TMVA` classifier configuration string (look up the `TMVA` documentation for details). The type of the classifier is given as *type* attribute. The list of supported classifiers is documented in the section 4, additional classifiers can be simply added by adding a line to the `switch` statement. Additionally, the *name* attribute has to be set. It specifies the name under which the classifier will appear in `TMVA`'s own monitoring `ROOT` file.

  Optionally a `setup` tag can be passed in order to control how the dataset is passed to `TMVA`. This tag has the two attributed *cuts* and *options*. The latter defaults to `SplitMode = Block:!V` and instructs TMVA to first run over all signal and then all background samples. For certain classifiers this is suboptimal and it might be a wise choice to switch to "random" mode instead. The *cuts* parameter is a `TreePlayer`-compatible cut string to select a subsample from the training sample.

5.6 CMSSW framework interaction modules and macros

The trainer provides a number of framework modules that have to be instantiated using the EventSetup record as template argument. Like the `MVAComputer`, the `PhysicsTools/MVAComputer/interface/HelperMacros.h` contains macros to instantiate those modules:

```
MVA_COMPUTER_CONTAINER_SAVE_IMPLEMENT(XYZRcd, XYZContainerSaveCondDB);
MVA_COMPUTER_FILE_SAVE_IMPLEMENT(XYZRcd, XYZSaveFile);
```

This macro will create an edm plugin named `XYZSaveFile` that allows to save a calibration object to the CondDB or standalone MVA files respectively. The objects are retrieved via EventSetup using the record `XYZRcd`. These objects are provided within the framework by an "ESSource", "ESProducer" or "ESProducerLooper". Using the file saver together with a source allows for conversion of calibration container objects (e.g. copying from CondDB to a file or vice versa), as well as storing the result of a training to a file.

The training within CMSSW is steered via an "ESProducerLooper". This is a special framework module that can instruct the framework's global event loop to repeatedly run over the EDM dataset, while updating the EventSetup objects that are produced at the same time, for each iteration. The "MVATrainerLooper" implements such a facility and ties the trainer iterations to the framework main loop and provides the training calibrations as well as the final calibration via the EventSetup mechanism. One of the calibration saver modules can then pick up the final calibration object and persistently store it.

```
MVA_TRAINER_CONTAINER_LOOPER_IMPLEMENT(XYZRcd, XYZTrainerLooper);
```

There is also a convenience macro that will instantiate the three modules as if the three macros were expanded manually, as given above:

```
MVA_TRAINER_IMPLEMENT(XYZ);
```

The corresponding MVA$_C$OMPUTER$_S$AVE$_I$MPLEMENTand

This section discusses the framework modules that the MVAComputer and MVATrainer provide. Note that since all the modules are templated with the EventSetup record as first argument and therefore explicitly have to be instantiated by the user and registered as EDM plugins.

5.7 Calibration object sources and savers

The MVAComputer provides templates for ''ESSources'' to read standalone MVA calibration object files from disk. Calibration objects can be read from the CMS Conditions Database using the builtin ''PoolDBESSource''. Only the modules that provide the PhysicsTools::Calibration::MVAComputerContainer container objects are discussed here, as the individual calibrations are deprecated.

The MVA$_C$OMPUTER$_C$ONTAINER$_F$ILE$_S$OURCE$_I$MPLEMENTmacrooranyoftheotherconveniencemacrossh

```
process.XYZFileSource = cms.ESSource("XYZFileSource",
        relativePath = cms.string('calib.mva'),
        absolutePath = cms.string('/some/other/path/calib2.mva'),
        fileInPath = cms.FileInPath('PhysicsTools/MVATrainer/test/calib3.mva')
)
```

Each of python keyword argument lines (i.e. the named parameters) adds one calibration object to the container. The name of the parameter is the label the calibration object is given within the container and the value is a path pointing to the calibration object file. This path can be either passed as a "cms.string" in which case it will be resolved with default unix semantics (i.e. starting from the local directory, or using the absolute path). Alternatively, it can be passed via "cms.FileInPath" in which case the file will be search for in the directories in the "CMSSW$_S$EARCH$_P$ATH"environmentvariable,which,amongothers,containsthe"src'directoryoftheCMSSWrel

The following lines retrieve a calibration object container from the CondDB:

```
from CondCore.DBCommon.CondDBSetup_cfi import *

process.XYZRcd = cms.ESSource("PoolDBESSource",
```

```
        CondDBSetup,
        timetype = cms.string('runnumber'),
        toGet = cms.VPSet(cms.PSet(
                record = cms.string('XYZRcd'),
                tag = cms.string('some_pooldb_tag')
        )),
        connect = cms.string('sqlite_file:localconditions.db'),
        BlobStreamerName = cms.untracked.string('TBufferBlobStreamingService')
)
```

Two parameters should be exchanged here, "some_pooldb_tag" is the CondDB tag under which the calibration container object is stored in the database, and "sqlite_file:localconditions.db" is the `POOL` URL to the database. This example uses a local SQLite file for the latter, "sqlite_fip:" can be used as well to pass the filename through the "FileInPath" mechanism. The official Conditions Database can be accessed via "frontier:" or "oracle:" (the exact paths need to be looked up in the documentation). Also, for production use the calibration objects are automatically retrieved via the global tag and the relevant sources are managed by the software calibration database integrators.

Also note that if the official database is used and a new record is created, a schema mapping has to be uploaded to the database before storing any data. Otherwise the automatic mapping is used, which creates too many individual SQL tables and subsequent retrieval of data from the database is heavily slowed down. A template schema mapping can be found in "CondFormats/PhysicsToolsObjects/xml/MVAComputerContainer_basic_0.xml", the record name has to be adjusted before use. The command line tool "pool_build_object_relational_mapping" can be used to upload the mapping to the database.

For saving calibration objects to disk, the corresponding module just works analoguosly:

```
process.XYZSaveFile = cms.EDAnalyzer("XYZSaveFile",
        testMVA = cms.string('testMVAComputerEvaluate.mva')
)
```

This line will cause the object with the label "testMVA" in the container object for the record "XYZRcd" to be saved into the given file. Note that the "FileInPath" mechanism cannot be used here. Saving multiple objects at the same time is possible by specifying multiple parameters, like for "XYZFileSource". The file saver additionally has an untracked boolean parameter named "trained", which defaults to true. This means that by default the module will try to retrieve the container object from the EventSetup with the "trained" label, as opposed to the empty label. In this mode it will attempt to save a trained calibration object from the `MVATrainerLooper` after the last iteration (i.e. the analyzer will wait for the calibration to appear), rather than a calibration container from one of the sources. By specifying this parameter and setting it to false, the latter can be achieved.

Storing an MVA calibration container object into the Conditions Database works similarly. It is composed of too parts: Loading the "PoolDBOutputService" and giving it the database coordinates and which tags to save, and the "XYZContainerSaveCondDB" analyzer that passes the object from the EventSetup to the "PoolDBOutputService".

```
process.PoolDBOutputService = cms.Service("PoolDBOutputService",
        BlobStreamerName = cms.untracked.string('TBufferBlobStreamingService'),
```

```
        DBParameters = cms.PSet( messageLevel = cms.untracked.int32(0) ),
        timetype = cms.untracked.string('runnumber'),
        connect = cms.string('sqlite_file:localconditions.db'),
        toPut = cms.VPSet(cms.PSet(
                record = cms.string('XYZRcd'),
                tag = cms.string('some_pooldb_tag')
        ))
)

process.XYZSave = cms.EDAnalyzer("XYZContainerSaveCondDB",
        toPut = cms.vstring(
                "justTrained",
                "justTrainedToo"
        ),
        toCopy = cms.vstring(
                "copyFromASource",
                "alsoCopyFromASource"
        )
)
```

The "PoolDBOutputService" service works analoguously to the "PoolDBESSource". The "toPut" parameter lists the records to store and assigns them the given CondDB tag. The "XYZContainerSaveCondDB" takes two lists of labels. The "toPut" list takes the calibration objects from the EventSetup with the "trained" label, i.e. from the trainer looper. The labels listed in "toCopy" are copied directly from respective EventSetup sources.

The trainer looper facilitates training inside the CMS software framework. One of the big advantages is that the training calibrations for each iteration are provided via EventSetup. This means that almost the same code can be used for passing the training dataset as for evaluation, the main difference being that for training the target information has to be provided as well. Hence, the training is done via EDM analyzers that retrieve the training calibration from the EventSetup, instantiate an MVAComputer and pass the variables. The `MVAComputerCache` can take over the recommended caching, as discussed in section 3. The trainer looper itself can instantiate one or more `MVATrainer` instances that are assigned to one or more object in the MVA calibration container. If multiple trainers are used, the training is performed in parallel.

```
process.looper = cms.Looper("XYZTrainerLooper",
        trainers = cms.VPSet(cms.PSet(
                calibrationRecord = cms.string('testMVA'),
                trainDescription = cms.untracked.string('testMVATrainer.xml'),
                loadState = cms.untracked.bool(False),
                saveState = cms.untracked.bool(False),
                monitoring = cms.untracked.bool(True)
        ))
)
```

The "trainers" parameter is a set of `PSet`'s, one per calibration object to be trained. The "calibrationRecord" refers to the label given within the container. "trainDescription" is the path to a filename (`cms.untracked.FileInPath` can be used as well). "loadState", "saveState" and "monitoring" refer to the respective boolean flags of the `MVATrainer` (the defaults if not

specified are all "false").

5.8 Using an Analyzer and MVATrainerLooper for Training

Writing an analyzer to feed data is almost identical to the way an analyer can use the `MVAComputer` for evaluation of a network:

```
class XYZTrainer : public edm::EDAnalyzer {
      ...
   private:
      PhysicsTools::MVAComputerCache mva;
};

void XYZTrainer::analyze(const edm::Event& iEvent,
                         const edm::EventSetup& iSetup)
{
   using namespace PhysicsTools;

   mva.update<XYZRcd>("trainer", iSetup, "testMVA");
   if (!mva)
      return;

   ...

   Variable::ValueList values;
   values.add(MVATrainer::kTargetId, target);
   // values.add(MVATrainer::kWeightId, 1.0);
   values.add("x", x);
   values.add("y", y);

   mva->eval(values);
}
```

One difference is the inserted first parameter `"trainer"` in the call to `MVAComputer::update<...>` which tells the `MVAComputerCache` to retrieve the calibration object from the EventSetup with the "trainer" label, i.e. explicitly request a training calibration from the `MVATrainerLooper`. The second difference is that the target information is provided in addition.

It is therefore strongly suggested to share as much code between the analyzer that performs the network evaluation and the analyzer that is used for training. For instance the filling of the variable list can be moved to a shared class, and the trainer just has to add the target information.

The general layout of a full CMSSW package, from the definition and registration of an EventSetup record, calibration storagen and retrieval plugins, an analyzer/producer evaluating the network and a training plugin, is somewhat complex and has to fulfill several criteria imposed by the build system, shared libraries and plugin rules. One possibility would be:

`.../interface/XYZRcd.h`
: define the record:
`MVA_COMPUTER_CONTAINER_DEFINE(XYZRcd);`

`.../interface/...`
: header for `Variable::ValueList` filling class

`.../src/XYZRcd.cc`
: register record with EventSetup:
`EVENTSETUP_RECORD_REG(XYZRcd);`

`.../src/...`
: source file for `Variable::ValueList` filling class

`.../plugins/XYZRcdPlugin.cc`
: define record plugin:
`MVA_COMPUTER_CONTAINER_RECORD_PLUGIN(XYZRcd);`

`.../plugins/XYZHelpers.cc`
: implement MVA calibration file ESSource plugin:
`MVA_COMPUTER_CONTAINER_FILE_SOURCE_IMPLEMENT( XYZRcd, XYZFileSource);`

`.../plugins/XYZAnalyzer.cc`
: the EDAnalyzer/EDProducer that evaluates the network

`.../plugins/XYZTrainer.cc`
: the EDAnalyzer to train the network

`.../plugins/XYZTrainerHelpers.cc`
: the looper, and CondDB / file save plugins:
`MVA_TRAINER_IMPLEMENT(XYZ);`

What is important here, is that code that is shared between both the analyzer and trainer code, is available in a common library (if the users of the common code are supposed to end up in separate plugins). In the proposal here, the `src` and `interface` directories are chosen for that purpose. As a consequence, the `BuildFile` on the top level of the package cannot declare the package as an EDM plugin (i.e. `<flags EDM_PLUGIN=1>`) and the plugins must be different in either a different package or the `plugins` subdirectory. Keeping the analyzer/producer that only evaluates the network separate from the trainer has the advantage that two different sets of dependencies can be used, as for network evaluation it is usually undesirable to depend on simulation and MC truth matching packages, which are typically needed to obtain the target information. In that directory, a number of plugins are then defined, like a plugin for the EventSetup record, a plugin for the evaluation and training analyzer, the trainer looper and the EventSetup read/store plugins. It is important to closely follow the separation for any EventSetup record related details, otherwise intransparent linking time or runtime errors will occur. Also, the analyzers have to be declared to the framework with the usual `DEFINE_FWK_MODULE(`*ClassName*`);`, which has not been explicitly listed so far.

In order to run a trainer, the CMSSW configuration needs:

- a trainer looper
- an analyzer to feed the trainer
- a calibration saver module
- a "PoolDBOutputService" in case the calibration is to be saved to the CondDB
- all other supporting modules, i.e. a source, eventually some producers, all relevant includes like services, conditions, geometry, and so on

All analyzers and calibration savers have to be put into a path, the looper just has to be defined and put into the process.

In addition to providing only the default trainer description file, it is usually of advantage to also provide a trainer description that employs the "TreeSaver" trainer processor to write all variables into a `ROOT` tree. This way, the machinery can be simply reused to dump all training data to a tree, which can then later be processed using the "mvaTreeTrainer", gaining a lot in turn-around time, since the heavy-weight framework is avoided. The resulting `.mva` calibration files can be transferred into the CondDB by running a dummy job containing only an `EmptySource`, `maxEvents` set to 1, an "ESSource" of type "XYZFileSource", a CondDB saver and a "PoolDBOutputSource".

6 The MVA Trainer Monitoring

This section discusses the monitoring feature that can be enable in the `MVATrainer`. It collects statistics from the trainer processors and the network and writes them out as collection of histograms to a `ROOT` file. The name of the file is determined by "trainfiles" option in the trainer description file, using the processor name "monitoring", e.g. "train_monitoring.root". This file can be accessed directly or inspected using the shipped ViewMonitoring `Cint` macro.

The contents of the monitoring files are organized in a flat hierarchy of subdirectories (`TDirectory`'s). The directory name is composed of the monitoring type and the trainer processor name, separated by an underscore. The monitoring type is either the name of the trainer processor (e.g. "ProcNormalize") or simply "input". The former contain processor-specific monitoring content and are only emitted if the trainer processor explicitly provides monitoring output. The latter is always created for all trainer processors and contains the distributions for the input variables. In addition, there is a directory "output" which contains the distributions for the final network discriminator.

The "input_..." directories contain two histograms for each input variable. The names of those histograms are constructed as "source_*name_sig/bkg*", with *source* and *name* referring to the input variable reference in the trainer processors "input" configuration section. The *sig* and *bkg* indicate whether the histogram contains data for signal or background. The histogramming is performed using `ROOT`'s automatic filling and on-the-fly binning, so the histogram ranges do not necessarily represent the minimum and maximum value of the respective variable. In case of variables with multiple appearances, all appearances are filled into the same histogram. The same scheme applies to the signal and background distribution of the network output in the "output" directory.

The processor-specific monitoring output is:

- **ProcNormalize** writes out the histograms for the pdf's, as they are stored in the calibration object. The naming scheme here is "source_*name*_pdf" or "*source_name*_CAT#_pdf" in case categories are used (# is the category index in that case).
- **ProcLikelihood** writes out the histograms for the pdf's, as they are stored in the calibration object. The naming scheme here is "$source\_name''_{sig/bkg}$ or "*source_name_CAT#_sig/bkg*" *incasecategoriesareused*.
- **ProcMatrix** writes out the symmetric correlation matrix as a two-dimensional histogram (`TH2F`) under the name "CorrMatrix". It contains one row/column per input variable, in order of their listing in the "input" section and additionally one row/column to store the

correlation to the target variable. The individual bins on the two axes are labelled using the scheme "source_*name*", the target variable is labelled "target". All bin contents are in the range $[-1, 1]$, the diagonal elements are set to 1.

In case the ranking feature is enabled, the directory contains an additional one-dimensional histogram named "Ranking" which contains the result of the ranking algorithm. It contains one bin per variable (labelled using the same scheme as the correlation matrix) that contains the correlation of the respective variable ensemble to the target, as described in section 5.5. The leftmost bin contains the whole variable ensemble. After removing the variable the bin is labelled with from the ensemble, one obtains the ensemble-to-target correlation in the adjacent bin on the right. The algorithm fills the histogram from the left to the right, so that the rightmost bins contain the highest-ranked variables.

6.1 The ViewMonitoring script

The `Cint` script "PhysicsTools/MVATrainer/test/ViewMonitoring.C" can be directly executed from the command line using;

```
root -l ViewMonitoring.C
```

It is not necessary to do this from the CMS software runtime environment, as it does not depend on any CMS software parts. By default it looks for the file named "train_monitoring.root" in the current directory. If another monitoring file is to be opened, one can specify it as a parameter using:

```
root -l ViewMonitoring.C'("filename.root")'
```

(Note that the double quotes around the filename argument need to be escaped from the shell.)

The script opens the file and displays a menu window that presents the user with the possibility to select the monitoring objects to present. When a monitoring object is selected, its contents are displayed in one or more canvasses, with up to six plots per window. The window contents are printed into files in parallel. These files can be found in the "plots/" subdirectory, which is created if necessary. The naming of the files is typically "plots/*type_name_plot.ext*", with *type_name* corresponding to the directory name in the monitoring file (see above), *name* the input variable name (if applicable), *plot* an arbitrary string depending on what the the plot contains and *ext* being the file name extension ("pdf", "eps" or "png"). By default, however, the summary mode is activated, which will only create a "plot/summary.ps" containing a hardcopy of the whole canvas windows that are shown. This can be changed using the "Options" button.

When clicking on one of the presented buttons, in most cases another window appears that allows to select a specific variable processor, for which the monitoring contents are to be presented.

- **Input Variables** allows to inspect the distributions of the input variables of a specific trainer processor. Note that these are not necessarily the global input variables, and depending on the network layout these can also have been processed by another variable processor. This is very useful to follow the variable transformation path through the network when stacked processors are employed. The signal and background distributions are shown as normalized and superimposed histograms.
- **Discriminator & Performance** shows the network output distributions and analyzes the network performance. Five plots are produced:

1. the normalized and distribution histograms for the network output
2. the cut efficiencies for signal and background: The whole discriminator range is scanned and for each possible working point the percentage of selected ($discr > working\ point$) signal and background events is determined. The results are shown as a two curves that range from 100% efficiency in the top left to 0% in the bottom right. The separation between signal and background is an indication for the discrimination power of the network. This diagram can be used to select the working point.
3. the efficiency-versus-purity curve (also known as "ROC" curve) shows the curve that arises when all working points are plotted in the efficiency-versus-purity dimension. Efficiency here is the "signal efficiency" and purity is 1 minus "background efficiency" (i.e. the percentage of background events wrongly classified as signal). The better the network, the closer the curve gets to the upper right corner.
4. the same curve in the "signal efficiency" versus "background efficiency" dimension, with the "background efficiency" being shown on a logarithmic scale: This representation is especially useful when having to choose a working point with a very high background rejection.
5. the discriminator-to-target correlation: This diagram shows the $S/(S+B)$ histograms for different discriminator values. A Bayes-like classifier should ideally show a linear behaviour, which means that the discriminator can be interpreted as a probability of the data point belonging to the signal category. In this case the correlation plot can be interpreted as a measure for the training quality.

- **ProcNormalize** shows the pdf's for all variables of the *ProcNormalize* processors, as stored in the calibration object.
- **ProcLikelihood (S, B)** shows the pdf's for signal and background for all variables of the *ProcLikelihood* processors.
- **ProcLikelihood (S / (S+B))** turns the separate signal and background pdf's of the *ProcLikelihood* processors into a combined $S/(S+B)$ representation for the individual variables.
- **ProcMatrix** shows the variable-variable and variable-target correlation matrix in a color-coded two-dimensional matrix. In case, the ranking feature is used, the highest-ranked ten variables, together with their respective ensemble-to-target correlation and individual contribution is shown as well.
- **Draw All** shows all plots at once. Beware, as this can potentially cause very many windows to be opened.
- **Options** allows the user to select the output format in which the plots are dumped to disk.
- **Quit** leaves `ROOT`.

7 Conclusions

References