CERN Summer Student Report

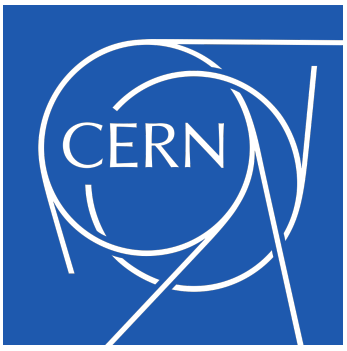
Modernization of the TMVA GUI

RVariablePlotter : modular plotting for TMVA

Simone Azeglio

Master Student, University of Turin
Visiting Student Researcher, University of Ottawa

Supervised by : Lorenzo Moneta, Sitong An
email : simone.azeglio@edu.unito.it



uOttawa

Contents

List of Figures

Introduction

The purpose of this report is to cover the work done during my Summer Student programme at CERN. It focuses on data visualization, which is an interdisciplinary field that deals with the graphical representation of information and data. In particular, this project focuses on the modernization of the TMVA Graphical User Inteface (GUI).

Data visualization is crucial for scientific modeling and for communicate information clearly. One of the prerequisite to obtain a satisfactory result, from this perspective, is to allow the toolkit of choice to be modular and not monolithic. In our work we pave the way for such a modularity by implementing a new class, namely *RVariablePlotter*, which provides tight integration with several ROOT's data structures, such as *RDataFrame* and *RTensor*, and allows plotting both feature's distributions and basic performance measurements for classification problems, e.g. *ROC Curves, ROC AUC Score*.

Along with this class, we provide several examples for an easier understanding, which can be further adapted to custom datasets or ROOT's output files. All the code has been submitted as a Pull Request (**PR #8723** [1]) on the official Github repository of the ROOT project.

1. https://github.com/root-project/root/pull/8723

Chapter 1

TMVA & Data

Introduction

THe TMVA library, part of the ROOT framework for large-scale data analysis developed at CERN, is dedicated to multivariate analysis, and in particular offers numerous machine learning algorithms in a standardized framework. It is widely used in High Energy Physics for data analysis, mainly to perform several tasks: from regression to classification and beyond. To keep up to date with the state of the art in deep learning, different deep learning modules have been developed.

TMVA's original paradigm is to book the methods through a Factory, that takes care of the data loading and sampling as well as feeding it properly to the given method. Several methods can be instantiated through the factory for a given task, trained, tested and evaluated at once. To keep up to date with the state of the art, a group of Google Summer of Code students has been implementing several deep learning modules for TMVA, offering a series of deep learning algorithms, including different Recurrent Neural Network architectures, e.g. LSTM, GRU and further integration with Pytorch and Keras, two widely known Python frameworks for Machine and Deep Learning.

Along with these developments, the necessity for a modern and modular GUI have risen (see PR #4211 [1]): the aim of this report is to illustrate the latest developments in this direction. Before delving into the details of our implementation, we should take a look at how data might be presented in the eyes of a ROOT user. In particular we are going to shed light onto two different data containers: *TTree* and their high-level interface *RDataFrame*, and *RTensor*.

1.1 TTree & RDataFrame

Typically, when you consider a problem that can be approached by applying Machine Learning techniques, data are coming in a tabular form: you have a dataset consisting of samples (usually rows) and features (usually columns), i.e. measured variables. A *TTree* [2] represents, in this view, a columnar dataset: i.e. a list of columns or branches. ROOT allows you to efficiently write to disk, in a compressed format, such data or keep them in memory up until a certain threshold is reached.

ROOT provides a high-level interface for accessing and working with such data stored in *TTree* or in specific files (e.g. CSV's): *RDataFrame* [3]. *RDataFrame* offers a highly optimized

1. https://github.com/root-project/root/pull/4211
2. For more details see official documentation at https://root.cern/doc/master/classTTree.html
3. See documentation and examples at https://root.cern/doc/master/classROOT_1_1RDataFrame.html

workflow: multithreading and different low level optimizations allow the user to exploit all the resources in an efficient manner. One of the most useful tricks, for example, consists in the possibility to apply transformations in a sequential fashion.

*RDataFrame*s are often encountered in data analysis pipeline [4]: that's why we started by implementing the *RVariablePlotter* class by specifically focusing on this. As we will show in Chapter 2, *RDataFrame* is our baseline data container, in the sense that our methods implicitly convert any other data container (e.g. *RTensor*) to a *RDataFrame* first and then plot results.

1.1.1 Example

Here it follows a simple example where we initialize two dataframes, by downloading a *.root* TMVA classification example from a specified url. The *.root* file is actually like a directory, we can look for the signal *TTree* and the background *TTree* by navigating into it. After the initialization, we can easily apply a transformation and define a new column, i.e. *var5* by multiplying *var1* and *var2*. In the end we create two new *RDataFrame*s with the new column.

```
// Initialize ROOT dataframes from signal and background datasets
   const std::string filename = "http://root.cern.ch/files/tmva_class_example.root
   ";
   ROOT::RDataFrame sig1("TreeS", filename);
   ROOT::RDataFrame bkg1("TreeB", filename);

   // Apply transformations on the datasets to be included in the study
   auto transform_ = [](ROOT::RDF::RNode df) { return df.Define("var5", "var1 *
   var2"); };
   auto sig2 = transform_(sig1);
   auto bkg2 = transform_(bkg1);

```

1.2 RTensor

As we have previously mentioned, data can come in form of an *RTensor* [5]. *RTensor* is a vector-like container with contigous memory and shape information. Its entries can be accessed by specifying indices, without caring about the intrinsic one-dimensional memory layout of the storage. Depending on the specific usage, *RTensor*s can both own the underlying contiguous memory or represent a view on existing data.

TMVA::Experimental's namespace provide also a method that can be proven to be useful for our purposes and consistency checks while converting a *RDataFrame* into a *RTensor* and vice-versa, namely: *AsTensor()* [6].

1.2.1 Example

We can try to transform our signal and background *RDataFrame*s in *RTensor* by employing the *AsTensor()* method, and take a look at how they differ by executing the code with ROOT.

4. For a quick hands-on practice crash course we refer the interested reader to the following content from the official documentation: https://root.cern/doc/master/classROOT_1_1RDataFrame.html#crash-course
5. See official documentation for more details at https://root.cern/doc/master/classTMVA_1_1Experimental_1_1RTensor.html
6. See https://root.cern/doc/master/namespaceTMVA_1_1Experimental.html#aad45f2c0d0e44b2b25714e665f8b803e

```
1
2     using namespace TMVA::Experimental;
3     // Initialize ROOT dataframes from signal and background datasets
4     const std::string filename = "http://root.cern.ch/files/tmva_class_example.root
      ";
5     ROOT::RDataFrame sig1("TreeS", filename);
6     ROOT::RDataFrame bkg1("TreeB", filename);
7
8     // Apply transformations on the datasets to be included in the study
9     auto transform_ = [](ROOT::RDF::RNode df) { return df.Define("var5", "var1 *
      var2"); };
10    auto sig2 = transform_(sig1);
11    auto bkg2 = transform_(bkg1);
12
13    // same data, we can transform them in tensors and compare
14    auto sig2Tensor = AsTensor<float>(sig2);
15    auto bkg2Tensor = AsTensor<float>(bkg2);
16
```

Chapter 2

Design and Implementation

Introduction

In Chapter 1 we have introduced typical ROOT's data containers, here on the other hand, we would like to detail *RVariablePlotter*'s implementation and its different plotting features. We start by illustrating different constructors, depending on which data container we have. After that we basically developed two complementary paths: the first one for *RDataFrame*s and the second for *RTensor*s. At this point, we illustrate two private methods that operate as *RTensor* to *RDataFrame* converter. Last but not least, we show how to apply these methods by employing a few examples.

Technical Specification: in this Chapter we have not considered a proper distinction between *RDataFrame*s and *RNode*s [1]. To be more specific, we actually use *RNode*s in the following code, which are essentially a more general type with the same functionality of *RDataFrame*s.

2.1 RVariablePlotter

2.1.1 Constructors

In the most general case, we would consider a vector of *RDataFrame*s as the input for our *RVariablePlotter*, along with some labels, specyfing each dataframe, for plotting multiple curves in the same figure (i.e. Canvas).

Constructor for a vector of *RDataFrame*s and labels

```
TMVA::RVariablePlotter::RVariablePlotter(const std::vector<ROOT::RDF::RNode>& nodes,
     const std::vector<std::string>& labels)
    : fNodes(nodes),
    fLabels(labels){

        if (fNodes.size() != fLabels.size())
            std::runtime_error("Number of given RDataFrame nodes does not match
    number of given class labels.");

        if (fNodes.size() == 0)
            std::runtime_error("Number of given RDataFrame nodes and number of given
     class labels cannot be zero.");
}
```

Since we might have data in the form of multiple *RTensor*s, we can define another constructor by employing constructors overloading in C++.

1. See official documentation at https://root.cern/doc/master/df025__RNode_8C.html

Constructor for a vector of *RTensor*s and labels

```
TMVA::RVariablePlotter::RVariablePlotter( const std::vector<TMVA::Experimental::
    RTensor<Float_t>>& tensors, const std::vector<std::string>& labels)
    : fTensors(tensors),
    fLabels(labels)
    {

        if (fTensors.size() != fLabels.size())
            std::runtime_error("Number of given RTensor components does not match
    number of given class labels.");

        if (fTensors.size() == 0)
            std::runtime_error("Number of given RTensor components and number of
    given class labels cannot be zero.");
}
}
```

2.1.2 TMVA Style vs. Custom Style

In order to produce plots that are consistent with previous TMVA's plots, we have implemented a method that can either set TMVA's default style or either a custom defined plotting style while keeping the existing Canvas.

```
void TMVA::RVariablePlotter::InitializeStyle(bool useTMVAStyle){
   // set custom style
   if (!useTMVAStyle) {
      gROOT->SetStyle("Plain");
      gStyle->SetOptStat(0);
      gPad->SetMargin(0.2, 0.9, 0.1, 0.9);
      gPad->SetGrid(1,1);
      return;
   }

   TMVA::TMVAGlob::SetTMVAStyle();
   gPad->SetMargin(0.2, 0.9, 0.1, 0.9);
}
```

2.1.3 Drawing - RDataFrame

Drawing Features Distributions

In order to draw a specific feature distribution (i.e. *variable*, the first argument of *TMVA::RVariablePlotter::Draw()*) from a vector of *RDataFrame*s, we firstly initialize the plotting style. After that we create a vector of *TH1D* [2], which are basically 1-D histograms, in order to store them for each *RDataFrame* and stack them by means of the next for loop (line 17). As a default, we specify the line color by making sure that variables coming from different dataframes have different colors (line 18). In the end, as a good practice we instantiate a clone of the original stack and we modify it by adding axis labels and a title.

```
void TMVA::RVariablePlotter::Draw(const std::string& variable, bool useTMVAStyle) {
   // Make histograms with TH1D
    TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);

    const auto size = fNodes.size();
```

2. See documentation https://root.cern/doc/master/classTH1D.html

```
    std::vector<ROOT::RDF::RResultPtr<TH1D>> histos;

    for (std::size_t i = 0; i < size; i++) {
        // Trigger event loop with computing the histogram
        ROOT::RDF::RResultPtr<TH1D> h = fNodes[i].Histo1D(variable);
        histos.push_back(h);
    }

   // Modify style and draw histograms
    THStack stack_;

    for (unsigned int i = 0; i < histos.size(); i++) {
        histos[i]->SetLineColor(i + 1);
        histos[i]->SetTitle(variable.c_str());
        histos[i]->SetStats(true);
        stack_.Add(histos[i].GetPtr());
   }

    auto clone = (THStack*) stack_.DrawClone("nostack");

    clone->SetTitle(variable.c_str());
    clone->GetXaxis()->SetTitle(variable.c_str());
    clone->GetYaxis()->SetTitle("Count");
    clone->GetYaxis()->SetTitleOffset( 1.50 );

}
```

Drawing Legend

In most plots, we might be interested in having a legend. In our case we designed a lightweight and indipendent method, i.e. *TMVA::RVariablPlotter::DrawLegend()*, which accept four float inputs, corresponding to vertices of the legend: in this way we can adapt it to each specific example.

```

void TMVA::RVariablePlotter::DrawLegend(float minX = 0.8, float minY = 0.8, float
    maxX = 0.9, float maxY = 0.9) {
    // make Legend from TLegend
    TLegend l(minX, minY, maxX, maxY);
    std::vector<TH1D> histos(fLabels.size());

    for (unsigned int i = 0; i < fLabels.size(); i++) {
        histos[i].SetLineColor(i + 1);
        l.AddEntry(&histos[i], fLabels[i].c_str(), "l");
    }
    l.SetBorderSize(1);
    l.DrawClone();
}
```

Drawing ROC Curve

Let's consider a typical classification task: we would like to be able to establish the goodness of our specific model. One of the most general performance metrics (in terms of dealing with unbalanced datasets as well) is the ROC curve (Receiver Operating Characteristic curve), and the related ROC AUC Score, i.e. the Area Under the ROC Curve. With *TMVA::RVariablePlotter::DrawROCCurve()* we can plot the ROC Curve for a single model (specified by the first argument, i.e. *output_variable*) along with the ROC AUC Score as a legend.

```
void TMVA::RVariablePlotter::DrawROCCurve(const std::string& output_variable , bool
    useTMVAStyle){

    // consistency check, i.e. we need signal vs background for the roc curve
    if (fNodes.size() != 2)
        std::runtime_error("In order to plot the ROC Curve you need 2 RNodes,
    corresponding to prediction on signal and background");

    // TMVA plotting style
    TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);

    auto sigDf = fNodes[0];
    auto bkgDf = fNodes[1];

    // extract columns as vectors of float for the selected model (i.e. specified by
     output_variable)
    auto sigModel = sigDf.Take<float>(output_variable).GetValue();
    auto bkgModel = bkgDf.Take<float>(output_variable).GetValue();

    // ROC Curve
    auto ROC = TMVA::ROCCurve(sigModel, bkgModel);
    auto ROCCurve = ROC.GetROCCurve();

    // ROC AUC Score as Legend's Label
    std::string AUC = std::to_string(ROC.GetROCIntegral());
    std::string legend = "AUC = ";
    const std::string legendStr = legend.append(AUC);
    const char *ROCAUCLabel = legendStr.c_str();

    // Graph title
    std::string varName = output_variable;
    std::string title = varName.append(" ROC Curve");

    auto clone = (TGraph*) ROCCurve;

    clone->SetTitle(title.c_str());
    clone->SetLineColor(1);
    // axis of the ROC Curve
    clone->GetXaxis()->SetTitle("Specificity");
    clone->GetYaxis()->SetTitle("Sensititvy");
    clone->GetYaxis()->SetTitleOffset(1.0);
    clone->DrawClone("AC");

    TLegend lROC(0.5, 0.8, 0.95, 0.9);
    lROC.AddEntry("clone", ROCAUCLabel, "lROC");
    lROC.DrawClone();

}
```

Drawing Multiple ROC Curves in a Single Figure

In many cases, we would like to have a single plot which compares ROC Curves for several trained models, along with ROC AUC Scores. We decided to design a specific method for this task: *TMVA::RVariablePlotter::DrawMultiROCCurve()*

```
void TMVA::RVariablePlotter::DrawMultiROCCurve(const std::vector<std::string>&
    output_variables, bool useTMVAStyle){

    // consistency check, i.e. we need signal vs background for the roc curve
    if (fNodes.size() != 2)
        std::runtime_error("In order to plot the ROC Curve you need 2 RNodes,
    corresponding to prediction on signal and background");
```

```
6
7     // TMVA plotting style
8     TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);
9
10    auto sigDf = fNodes[0];
11    auto bkgDf = fNodes[1];
12
13    // here we can store signals and backgrounds from each model
14    std::vector<std::vector<float>> sigModels;
15    std::vector<std::vector<float>> bkgModels;
16
17    for (unsigned int i = 0; i < output_variables.size(); i++){
18        // extract columns as vectors of float for the selected model (i.e.
      specified by output_variable)
19        sigModels.push_back(sigDf.Take<float>(output_variables[i]).GetValue());
20        bkgModels.push_back(bkgDf.Take<float>(output_variables[i]).GetValue());
21    }
22
23   std::string title = "ROC Curves";
24
25    // ROC first curve
26    auto ROC = TMVA::ROCCurve(sigModels[0], bkgModels[0]);
27    // Label = AUC Score for each model as sequence of char
28    std::string AUC = std::to_string(ROC.GetROCIntegral());
29    std::string legend = " AUC = ";
30    std::string legendStr = legend.append(AUC);
31    std::string out_var = output_variables[0];
32    std::string labelName = out_var.append(legendStr);
33    const char *ROCAUCLabel = labelName.c_str();
34
35    auto clone = (TGraph*) ROC.GetROCCurve();
36    clone->SetTitle(title.c_str());
37    clone->SetLineColor(1);
38    // axis of the ROC Curve
39    clone->GetXaxis()->SetTitle("Specificity");
40    clone->GetYaxis()->SetTitle("Sensititvy");
41    clone->GetYaxis()->SetTitleOffset(1.0);
42    clone->DrawClone("AC");
43
44    // Legend
45    auto lROC = new TLegend(0.65, 0.75, 0.90, 0.90);
46    lROC->AddEntry(clone, ROCAUCLabel, "l");
47    lROC->DrawClone();
48
49    for (unsigned int i = 1; i < sigModels.size(); i++){
50        // ROC instance
51        auto ROC = TMVA::ROCCurve(sigModels[i], bkgModels[i]);
52        // Label = AUC Score for each model as sequence of char
53        std::string AUC = std::to_string(ROC.GetROCIntegral());
54        std::string legend = " AUC = ";
55        std::string legendStr = legend.append(AUC);
56        std::string out_var = output_variables[i];
57        std::string labelName = out_var.append(legendStr);
58        const char *ROCAUCLabel = labelName.c_str();
59
60        auto clone = (TGraph*) ROC.GetROCCurve();
61        clone->SetLineColor(i + 1);
62        clone->DrawClone("CP");
63        lROC->AddEntry(clone, ROCAUCLabel, "l");
64        lROC->DrawClone();
65    }
66
```

```
}
```

2.1.4 Private Methods

RTensor to RDataFrame Converter - (Static Method)

So far, we have only dealt with *RDataFrame*s: in order to procede with *RTensor*s, we have implemented two private methods to allow for *RTensor* to *RDataFrame* conversion and for converting vectors of *RTensor*s to vectors of *RDataFrame*s in order to exploit the previously developed methods for plotting.

In *TMVA::RVariablePlotter::TensorToNode()* we convert a single *RTensor* into a *RDataFrame*. In the current implementation, while converting we employ a lambda function which couldn't work by referencing original *RTensor*'s columns: by copying *RTensor*'s entries we might experience slow execution whenever we have large tensors. Solving this issue by correctly referencing instead of copying could be one of the next improvements.

```
ROOT::RDF::RNode TMVA::RVariablePlotter::TensorToNode(const TMVA::Experimental::
    RTensor<Float_t>& tensor,  const std::vector<std::string>& variables){

    // shape check
    if (tensor.GetShape().size() != 2)
        std::runtime_error("Number of given RTensor dimensions does not match number
     of given class labels.");

    auto dfSig = ROOT::RDataFrame(tensor.GetShape()[0]).DefineSlotEntry(variables
    [0], [=] (unsigned, ULong64_t entry) { return tensor(entry, 0); } );

    std::size_t nvar = variables.size();
    // in case RTensors have a different number of columns
    if(tensor.GetShape()[1]!= variables.size())
        nvar = std::min(tensor.GetShape()[1], variables.size());

    for (std::size_t j = 1; j < nvar; j++){
        dfSig = dfSig.DefineSlotEntry(variables[j],  [=] (unsigned, ULong64_t entry)
     { return tensor(entry,j);});
            }

    // RDataFrame to RNode
    auto DFNode = ROOT::RDF::RNode(dfSig);

    return DFNode;
}
```

RTensors vector to RDataFrames vector converter

In *TMVA::RVariablePlotter::TensorsToNodes()* we iterate the previously analysed method, i.e. *TMVA::RVariablePlotter::TensorToNode()*, in order to convert a vector of *RTensor*s into a vector of *RDataFrame*s.

```
std::vector<ROOT::RDF::RNode> TMVA::RVariablePlotter::TensorsToNodes(const std::
    vector<std::string>& variables){

    const auto size = fTensors.size();
    std::vector<ROOT::RDF::RNode> RNodeVec;

    // loop through tensors
    for (std::size_t k = 0; k < size; k++){
        RNodeVec.push_back(TensorToNode(fTensors[k], variables));
```

```
    }

    return RNodeVec;
}
```

2.1.5 Drawing - RTensor

Drawing Features Distributions

Now that we have outlined how to convert *RTensor*s to *RDataFrame*s we can finally implement a method for plotting features distributions, which is similar to *TMVA::RVariablePlotter::Draw()* in design, the only difference is the number of arguments: we need to pass the name of all the features (i.e. the third argument, *variables*) along with the single one that we want to plot.

```

void TMVA::RVariablePlotter::DrawTensor(const std::string& variable, const std::
    vector<std::string>& variables, bool useTMVAStyle) {

    // vector of RNodes
    auto RNodeVec = TMVA::RVariablePlotter::TensorsToNodes(variables);

    TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);

    // Make histograms with TH1D
    const auto size = RNodeVec.size();
    std::vector<ROOT::RDF::RResultPtr<TH1D>> histos;

    for (std::size_t i = 0; i < size; i++) {
        // Trigger event loop with computing the histogram
        ROOT::RDF::RResultPtr<TH1D> h = RNodeVec[i].Histo1D(variable);
        histos.push_back(h);
        h = ROOT::RDF::RResultPtr<TH1D>();
    }

   // Modify style and draw histograms
    THStack stack;

    for (unsigned int i = 0; i < histos.size(); i++) {
        histos[i]->SetLineColor(i + 1);
        histos[i]->SetTitle(variable.c_str());
        histos[i]->SetStats(true);
        stack.Add(histos[i].GetPtr());
   }

    auto clone = (THStack*) stack.DrawClone("nostack");

    clone->SetTitle(variable.c_str());
    clone->GetXaxis()->SetTitle(variable.c_str());
    clone->GetYaxis()->SetTitle("Count");
    clone->GetYaxis()->SetTitleOffset( 1.50 );

}
```

Drawing ROC Curve

In a similar way, we employed the converter again for ROC Curves from *RTensor*'s data.

```
void TMVA::RVariablePlotter::DrawROCCurveTensor(const std::string& output_variable,
    const std::vector<std::string>& output_variables, bool useTMVAStyle){

    // consistency check, i.e. we need signal vs background for the roc curve
```

```
    if (fTensors.size() != 2)
        std::runtime_error("In order to plot the ROC Curve you need 2 RTensors,
    corresponding to prediction on signal and background");

    auto RNodeVec = TensorsToNodes(output_variables);

    // TMVA plotting style
    TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);

    auto sigDf = RNodeVec[0];
    auto bkgDf = RNodeVec[1];

    // extract columns as vectors of float for the selected model (i.e. specified by
     output_variable)
    auto sigModel = sigDf.Take<float>(output_variable).GetValue();
    auto bkgModel = bkgDf.Take<float>(output_variable).GetValue();

    // ROC Curve
    auto ROC = TMVA::ROCCurve(sigModel, bkgModel);
    auto ROCCurve = ROC.GetROCCurve();

    // ROC AUC Score as Legend's Label
    std::string AUC = std::to_string(ROC.GetROCIntegral());
    std::string legend = "AUC = ";
    const std::string legendStr = legend.append(AUC);
    const char *ROCAUCLabel = legendStr.c_str();

    // Graph title
    std::string varName = output_variable;
    std::string title = varName.append(" ROC Curve");

    auto clone = (TGraph*) ROCCurve;

    clone->SetTitle(title.c_str());
    clone->SetLineColor(1);
    // axis of the ROC Curve
    clone->GetXaxis()->SetTitle("Specificity");
    clone->GetYaxis()->SetTitle("Sensititvy");
    clone->GetYaxis()->SetTitleOffset(1.0);
    clone-> DrawClone("AC");

    TLegend lROC(0.5, 0.8, 0.95, 0.9);
    lROC.AddEntry("clone", ROCAUCLabel, "lROC");
    lROC.DrawClone();

}
```

Drawing Multiple ROC Curves in a Single Figure

Again, we used the converter for plotting ROC Curves in the same figure from *RTensor*'s data.

```

void TMVA::RVariablePlotter::DrawMultiROCCurveTensor(const std::vector<std::string>&
     output_variables, bool useTMVAStyle){

    // consistency check, i.e. we need signal vs background for the roc curve
    if (fTensors.size() != 2)
        std::runtime_error("In order to plot the ROC Curve you need 2 RTensors,
    corresponding to prediction on signal and background");

    auto RNodeVec = TensorsToNodes(output_variables);
```

```
9
10    // TMVA plotting style
11    TMVA::RVariablePlotter::InitializeStyle(useTMVAStyle);
12
13    auto sigDf = RNodeVec[0];
14    auto bkgDf = RNodeVec[1];
15
16    // here we can store signals and backgrounds from each model
17    std::vector<std::vector<float>> sigModels;
18    std::vector<std::vector<float>> bkgModels;
19
20    for (unsigned int i = 0; i < output_variables.size(); i++){
21        // extract columns as vectors of float for the selected model (i.e.
      specified by output_variable)
22        sigModels.push_back(sigDf.Take<float>(output_variables[i]).GetValue());
23        bkgModels.push_back(bkgDf.Take<float>(output_variables[i]).GetValue());
24    }
25
26   std::string title = "ROC Curves";
27
28    // ROC first curve
29    auto ROC = TMVA::ROCCurve(sigModels[0], bkgModels[0]);
30    // Label = AUC Score for each model as sequence of char
31    std::string AUC = std::to_string(ROC.GetROCIntegral());
32    std::string legend = " AUC = ";
33    std::string legendStr = legend.append(AUC);
34    std::string out_var = output_variables[0];
35    std::string labelName = out_var.append(legendStr);
36    const char *ROCAUCLabel = labelName.c_str();
37
38    auto clone = (TGraph*) ROC.GetROCCurve();
39    clone->SetTitle(title.c_str());
40    clone->SetLineColor(1);
41    // axis of the ROC Curve
42    clone->GetXaxis()->SetTitle("Specificity");
43    clone->GetYaxis()->SetTitle("Sensititvy");
44    clone->GetYaxis()->SetTitleOffset(1.0);
45    clone->DrawClone("AC");
46
47    // Legend
48    auto lROC = new TLegend(0.65, 0.75, 0.90, 0.90);
49    lROC->AddEntry(clone, ROCAUCLabel, "l");
50    lROC->DrawClone();
51
52    for (unsigned int i = 1; i < sigModels.size(); i++){
53        // ROC instance
54        auto ROC = TMVA::ROCCurve(sigModels[i], bkgModels[i]);
55        // Label = AUC Score for each model as sequence of char
56        std::string AUC = std::to_string(ROC.GetROCIntegral());
57        std::string legend = " AUC = ";
58        std::string legendStr = legend.append(AUC);
59        std::string out_var = output_variables[i];
60        std::string labelName = out_var.append(legendStr);
61        const char *ROCAUCLabel = labelName.c_str();
62
63        auto clone = (TGraph*) ROC.GetROCCurve();
64        clone->SetLineColor(i + 1);
65        clone->DrawClone("CP");
66        lROC->AddEntry(clone, ROCAUCLabel, "l");
67        lROC->DrawClone();
68    }
69
```

```
}
```

2.2 Examples

In this section, we provide various examples [3] of the previously described methods.

2.2.1 TMVA::RVariablePlotter::Draw()

```
void tmva005_RVariablePlotter()
{
    // Initialize ROOT dataframes from signal and background datasets
    const std::string filename = "http://root.cern.ch/files/tmva_class_example.root
    ";
    ROOT::RDataFrame sig1("TreeS", filename);
    ROOT::RDataFrame bkg1("TreeB", filename);

    // Apply transformations on the datasets to be included in the study
    auto transform_ = [](ROOT::RDF::RNode df) { return df.Define("var5", "var1 *
    var2"); };
    auto sig2 = transform_(sig1);
    auto bkg2 = transform_(bkg1);

    // Create a variable plotter object giving the dataframes and the class labels.
    TMVA::RVariablePlotter plotter({sig2, bkg2}, {"Signal", "Background"});

    TCanvas *c = new TCanvas("c", "c", 1400, 800);
    c->Divide(3, 2);

    // legend vertices
    float minX = 0.7;
    float minY = 0.8;
    float maxX = 0.9;
    float maxY = 0.9;


    // Place plots on the pads of the canvas
    const std::vector<std::string> vars = {"var1", "var2", "var3", "var4", "var5"};

    for (unsigned int i = 0; i < vars.size(); i++) {
        c->cd(i + 1);
        gPad->SetMargin(0.2, 0.9, 0.1, 0.9);
        c->Update();
        //gPad->SetGrid(1,1); // plotting a background grid
        plotter.Draw(vars[i], true);
        plotter.DrawLegend(minX, minY, maxX, maxY);


    }
```

The output of this macro can be observed in Figure 2.1.

2.2.2 TMVA::RVariablePlotter::DrawTensor()

```

void tmva006_RVariablePlotter_RTensor()
```

3. The code for these (and more) examples can be found in the PR https://github.com/root-project/root/pull/8723 as well

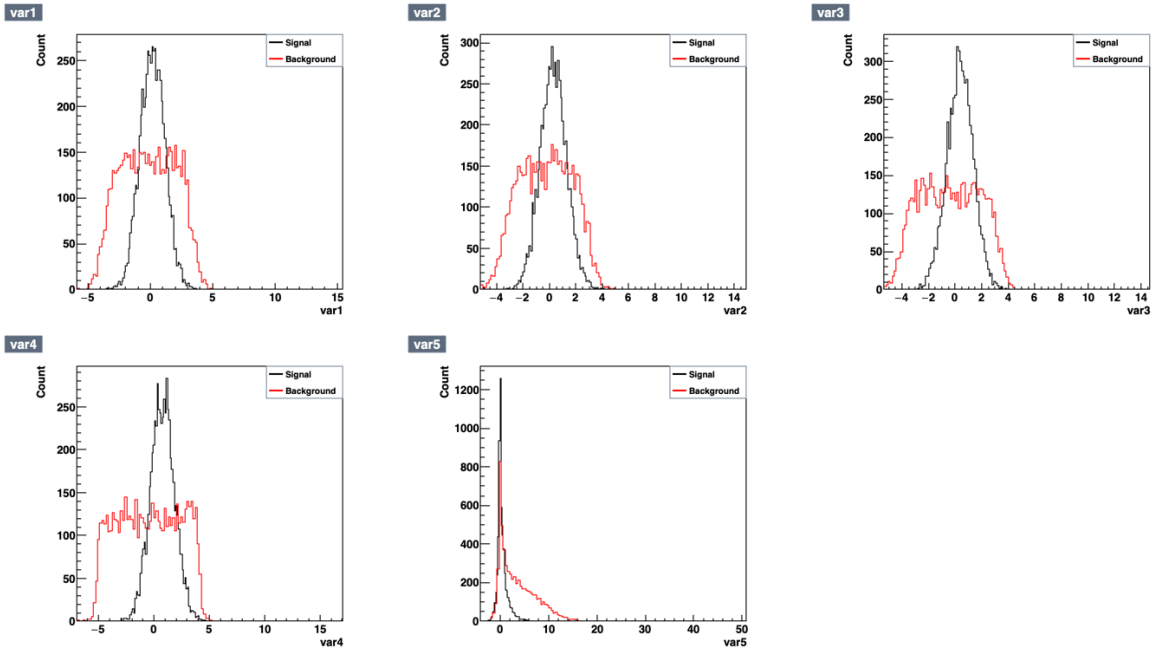


Figure 2.1 – Feature distributions, Signal vs Background. Taken from example *tmva005_RVariablePlotter.C*

```
{
    // Initialize ROOT dataframes from signal and background datasets
    const std::string filename = "http://root.cern.ch/files/tmva_class_example.root
    ";
    ROOT::RDataFrame sig1("TreeS", filename);
    ROOT::RDataFrame bkg1("TreeB", filename);

    // Apply transformations on the datasets to be included in the study
    auto transform_ = [](ROOT::RDF::RNode df) { return df.Define("var5", "var1 *
    var2"); };
    auto sig2 = transform_(sig1);
    auto bkg2 = transform_(bkg1);


    // same data as in tmva005.... , we can transform them in tensors and compare
    the plots
    auto sig2Tensor = AsTensor<float>(sig2);
    auto bkg2Tensor = AsTensor<float>(bkg2);

    // Place plots on the pads of the canvas - new columns are always added first
    //we should remember it while converting an RTensor to an RNode
    const std::vector<std::string> vars = sig2.GetColumnNames(); //{"var5", "var2",
    "var3", "var4", "var1"};

    // Create a variable plotter object giving the Tensors and the class labels.
    TMVA::RVariablePlotter plotter({sig2Tensor, bkg2Tensor}, {"Signal", "Background
    "});


    TCanvas *c = new TCanvas("c", "c", 1400, 800);
    c->Divide(3, 2);

```

```
    // legend vertices
    float minX = 0.7;
    float minY = 0.8;
    float maxX = 0.9;
    float maxY = 0.9;

    for (unsigned int i = 0; i < vars.size(); i++) {
        c->cd(i + 1);
        gPad->SetMargin(0.2, 0.9, 0.1, 0.9);
        c->Update();
        //gPad->SetGrid(1,1); // plotting a background grid
        plotter.DrawTensor(vars[i], vars, true);
        plotter.DrawLegend(minX, minY, maxX, maxY);
    }

}
```

The output of this macro can be observed in Figure 2.2. It eventually returns the same result as Figure 2.1 with a different order of single variables, due to the fact that while using *AsTensor()* we lose the variables names.

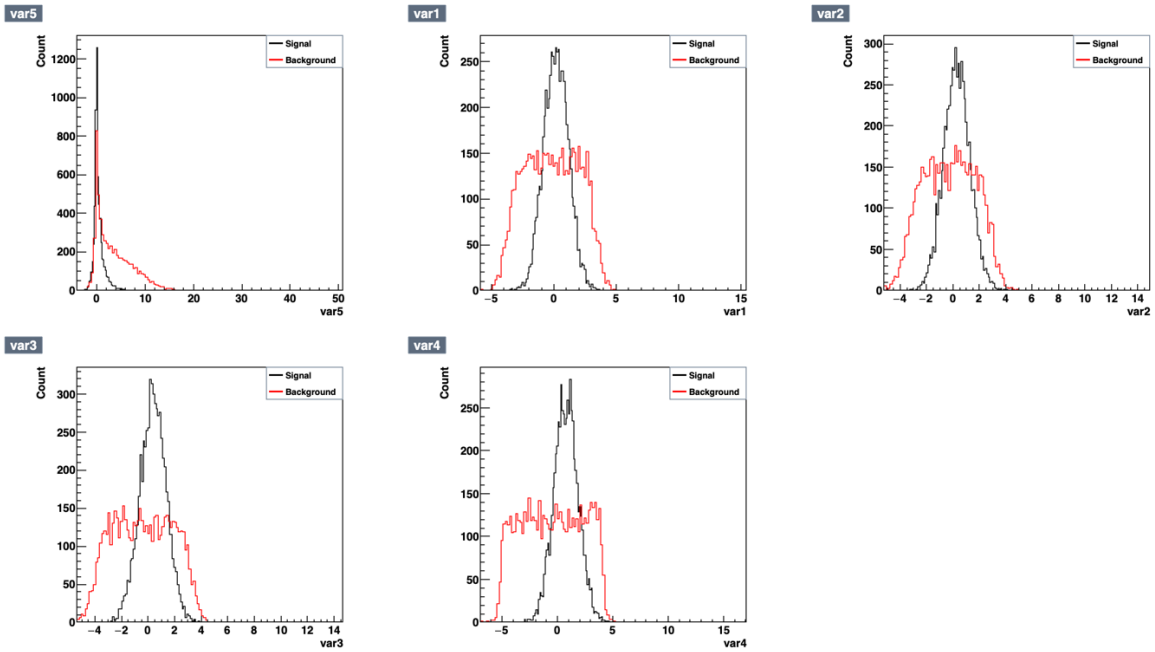


Figure 2.2 – Feature distributions, Signal vs Background. Taken from example *tmva006_RVariablePlotter_RTensor.C*

Plotting from model's RTensor output

```
// from tmva003_RReader.C
void train(const std::string &filename)
{
    // Create factory
    auto output = TFile::Open("TMVA.root", "RECREATE");
    auto factory = new TMVA::Factory("tmva003",
            output, "!V:!DrawProgressBar:AnalysisType=Classification");

    // Open trees with signal and background events
```

```
    auto data = TFile::Open(filename.c_str());
    auto signal = (TTree *)data->Get("TreeS");
    auto background = (TTree *)data->Get("TreeB");

    // Add variables and register the trees with the dataloader
    auto dataloader = new TMVA::DataLoader("tmva003_BDT");
    const std::vector<std::string> vars = {"var1", "var2", "var3", "var4"};
    for (const auto &var : vars) {
        dataloader->AddVariable(var);
    }
    dataloader->AddSignalTree(signal, 1.0);
    dataloader->AddBackgroundTree(background, 1.0);
    dataloader->PrepareTrainingAndTestTree("", "");

    // Train a TMVA method
    factory->BookMethod(dataloader, TMVA::Types::kBDT, "BDT", "!V:!H:NTrees=300:
    MaxDepth=2");
    factory->TrainAllMethods();
}


\subsection{TMVA::RVariablePlotter::DrawROCCurve()}

\begin{lstlisting}
void tmva009_RVariablePlotter_Higgs_ROC()
{
    // Initialize ROOT dataframes from signal and background datasets
    const std::string filename = "Higgs_ClassificationOutput.root";
    ROOT::RDataFrame testDf("dataset/TestTree", filename);

    // extracting signal & background
    auto sigDf  = testDf.Filter("classID==0");
    auto bkgDf  = testDf.Filter("classID==1");

    // columns (you can plot the ones you prefer)
    // const std::vector<std::string> vars = sigDf.GetColumnNames();
    // { "classID", "className", "m_jj", "m_jjj", "m_lv", "m_jlv",
    // "m_bb", "m_wbb", "m_wwbb", "weight", "BDT", "DNN_CPU", "Fisher", "Likelihood
    ", "prob_Fisher" }

    // for example
    const std::vector<std::string> vars = {"BDT", "DNN_CPU", "Fisher", "Likelihood
    "};

    // Create a variable plotter object giving the Tensors and the class labels.
    TMVA::RVariablePlotter plotter({sigDf, bkgDf}, {"Signal", "Background"});

    TCanvas *c = new TCanvas("c", "c", 1400, 800);
    c->Divide(3, 2);

    // use tmva style in DrawROCCurve() or custom style
    bool useTMVAStyle = true;

    for (unsigned int i = 0; i < vars.size(); i++) {
        c->cd(i + 1);
        c->Update();
        plotter.DrawROCCurve(vars[i], useTMVAStyle);

    }

}
```

The output of this macro can be observed in Figure 2.3.

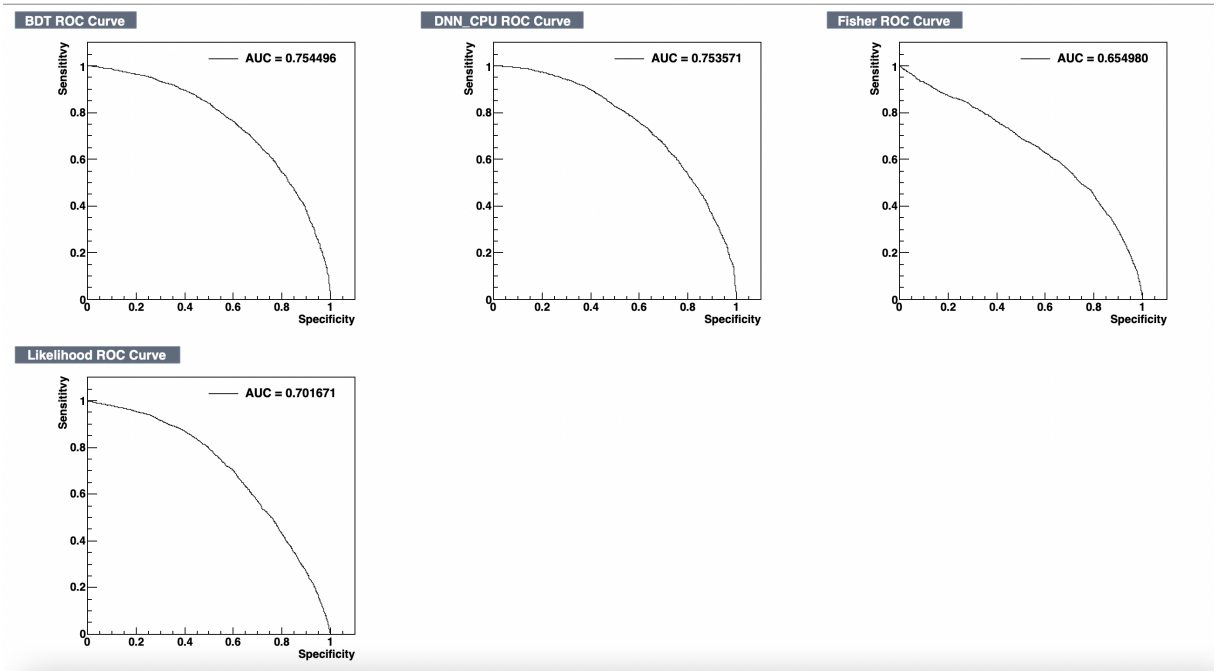


Figure 2.3 – ROC Curves from several trained models, taken from *tmva009_RVariablePlotter_Higgs_ROC.C*

2.2.3 TMVA::RVariablePlotter::DrawMultiROCCurve()

```
void tmva010_RVariablePlotter_Higgs_MultiROC()
{
    // Initialize ROOT dataframes from signal and background datasets
    const std::string filename = "Higgs_ClassificationOutput.root";
    ROOT::RDataFrame testDf("dataset/TestTree", filename);

    // extracting signal & background
    auto sigDf  = testDf.Filter("classID==0");
    auto bkgDf  = testDf.Filter("classID==1");


    // some of the output variables
    const std::vector<std::string> vars = {"BDT", "DNN_CPU", "Fisher", "Likelihood
    "};

    // Create a variable plotter object giving the Tensors and the class labels.
    TMVA::RVariablePlotter plotter({sigDf, bkgDf}, {"Signal", "Background"});

    TCanvas *c = new TCanvas("c", "c", 900, 600);

    // use tmva style in DrawMultiROCCurve() or custom style
    bool useTMVAStyle = false;

    plotter.DrawMultiROCCurve(vars, useTMVAStyle);

}
```

The output of this macro can be observed in Figure 2.4, which is basically the aggregated plot of the four curves in the previous example.

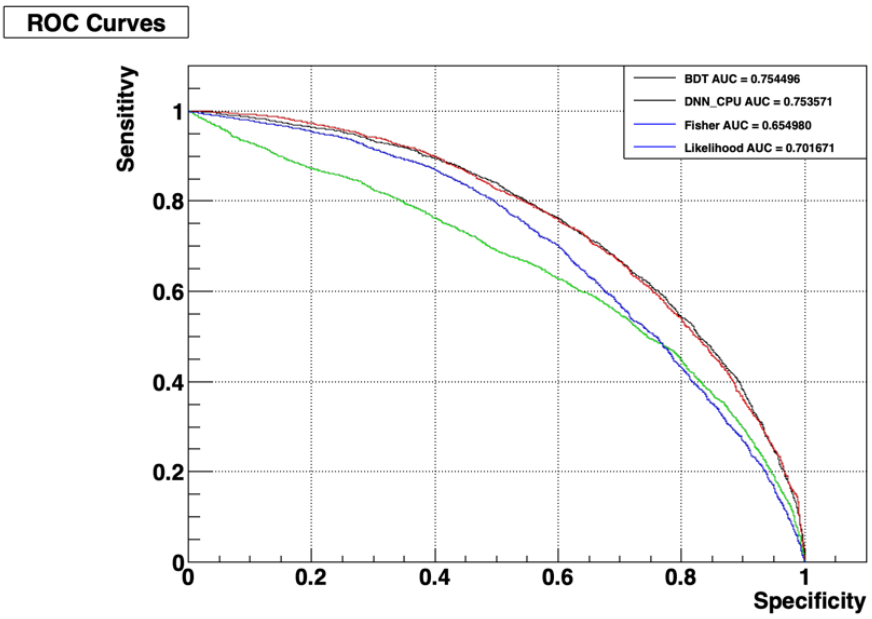


Figure 2.4 – Multiple ROC Curves in a single plot, taken from *tmva010_RVariablePlotter_Higgs_MultiROC.C*

Chapter 3

Conclusions

In this project, we started formalizing a proof of concept for an early TMVA GUI replacement. The examples demonstrate that a modular plotting interface is simple and intuitive. So far there is not yet a large number of methods, but starting from *RVariablePlotter*, adding more functionalities should be straightforward. Besides the research activity and implementation details, this project allowed me to take an insight in ROOT's development, and especially in TMVA, the machine learning library. I learned a lot about ROOT, data analysis, and low level implementations of Machine Learning algorithms by exploring already existing examples and source code. I also improved my software engineering skills, by revising C++ and by using Git, CMake, and working on such an open-source and large-scale collaborative project. This experience will be very valuable for my future as a physicist, especially with respect to my knowledge in computing. I also enjoyed the social experience, e-meeting a lot of people from all over the world, and CERN's open and friendly atmosphere which can be sensed even virtually.

Acknowledgements

The realization of this project would not have been possible without the help of some people. In this perspective, I would like to thank Lorenzo Moneta, PhD and Dr. Sitong An who gave me the opportunity to work at CERN and supported me during the project. I would also like to thank Dr. Omar Andres Zapata Mesa, who was always available for questions or discussions whenever I was stuck on something, and with whom it was friendly and enjoyable to work. Last but not least, I would like to thank all the fellow Summer Students and GSoC Students part of the TMVA team: Xandru Mifsud, Federico Sossai, Afan Terkom Sanjiban Sengupta, Aaradhya Saxena and Ahmat Hamdan.

And I would also like to thank the entire ROOT group for having been so nice colleagues during this Summer Student programme.