

2016 Exam Q5

G. Lowry
RHUL Physics

~~Y1, Y2~~ $y_i \sim \text{Gauss}(\lambda, \sigma_i), i=1,2$

↳ indep.

a) $\Rightarrow L(\lambda) = \prod_{i=1}^2 \frac{1}{\sqrt{2\pi} \sigma_i} e^{-\frac{(y_i - \lambda)^2}{2\sigma_i^2}}$

$\Rightarrow \ln L(\lambda) = -\frac{1}{2} \sum_{i=1}^2 \frac{(y_i - \lambda)^2}{\sigma_i^2} + C$

\Rightarrow minimize $\chi^2(\lambda) = \sum_{i=1}^2 \frac{(y_i - \lambda)^2}{\sigma_i^2}$

b) $\frac{d\chi^2}{d\lambda} = -\frac{2(y_1 - \lambda)}{\sigma_1^2} + -\frac{2(y_2 - \lambda)}{\sigma_2^2} \stackrel{\text{set}}{=} 0$

$\Rightarrow \hat{\lambda} = \frac{y_1/\sigma_1^2 + y_2/\sigma_2^2}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}$

$w_1 = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$

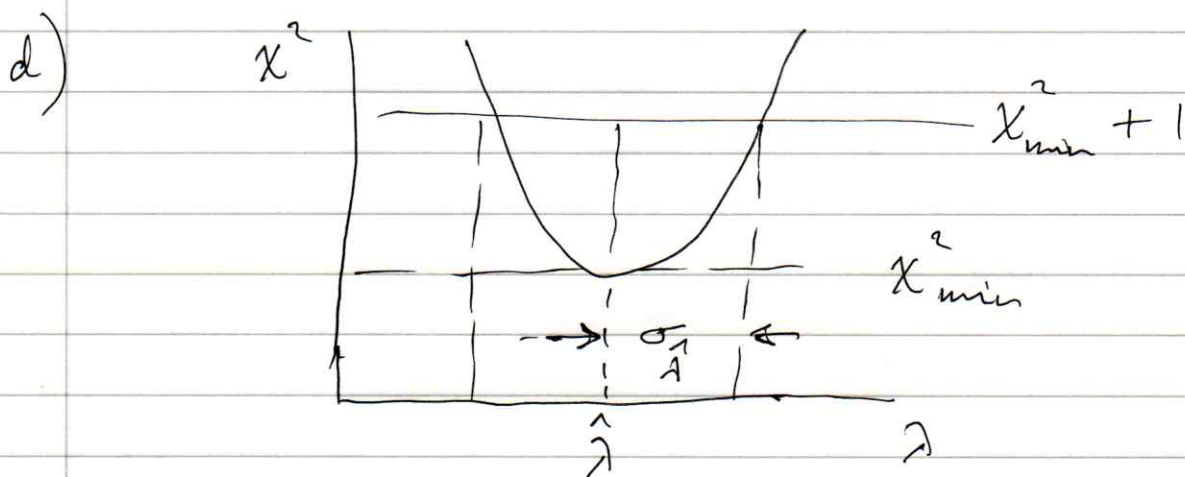
$\left(\frac{\sigma_2^2 y_1 + \sigma_1^2 y_2}{\sigma_1^2 + \sigma_2^2} \right)$

c) $E[\hat{\lambda}] = \frac{E[y_1]/\sigma_1^2 + E[y_2]/\sigma_2^2}{1/\sigma_1^2 + 1/\sigma_2^2}$

$= \lambda$ ✓ unbiased.

$$V[\hat{\lambda}] = \frac{1}{\left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}\right)^2} \left(\frac{V[y_1]}{\sigma_1^4} + \frac{V[y_2]}{\sigma_2^4} \right)$$

$$= \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}$$



$$p = \int_{x_{\min}}^{\infty} f_{x^2}(t; n_d = 1) dt$$

e) Suppose $\hat{\lambda} = w y_1 + (1-w) y_2$

= most general linear unbiased $\hat{\lambda}$

$$E[\hat{\lambda}] = w E[y_1] + (1-w) E[y_2] = \lambda$$

$$V[\hat{\lambda}] = w^2 \sigma_1^2 + (1-w)^2 \sigma_2^2$$

$$\frac{\partial V[\hat{\lambda}]}{\partial w} = 2w \sigma_1^2 - 2(1-w) \sigma_2^2 = 0$$

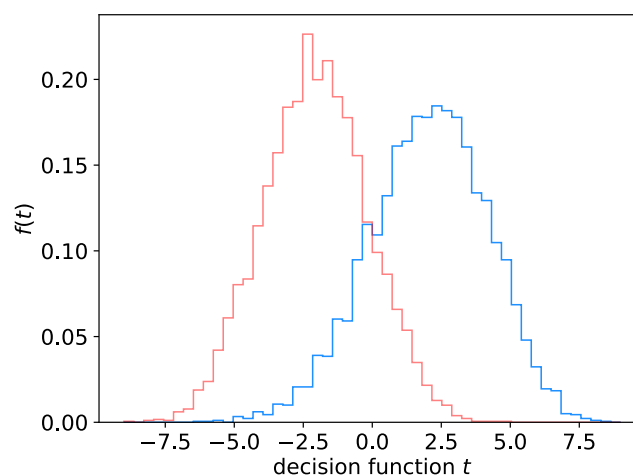
$$\Rightarrow w = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \Rightarrow \text{same as LS} \Rightarrow \text{"BLUE"}$$

Question 1, python version: First, the line of code

```
X = X[:,0:2]
```

was removed so that the classifiers use all three components of the feature vector (the code that makes the scatterplot was also sidestepped).

For the linear discriminant, the program as given already produced the following distribution for the decision function (not necessary to submit):



1(a) [5 marks] To calculate the power, add

```
power = metrics.accuracy_score(y_sig_test, y_sig_pred)
```

1(b) [5 marks] To calculate the size, add

```
size = 1. - metrics.accuracy_score(y_bkg_test, y_bkg_pred)
```

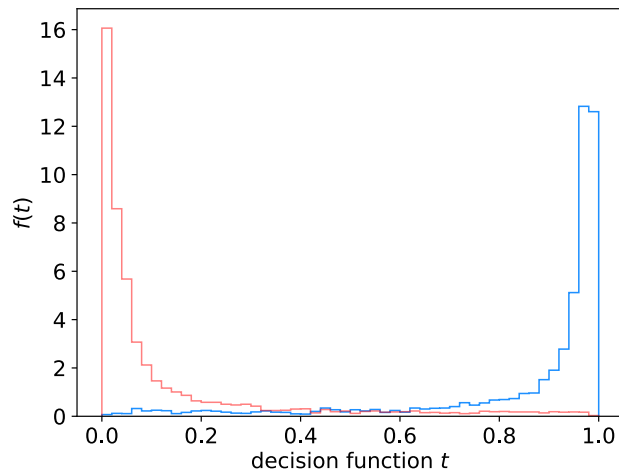
Combined with the existing code to find the test's size, the output for the LDA was:

```
power of test with respect to signal      = 0.8245684464070654
size of test of background                 = 0.14129135113591074
purity of signal sample                   = 0.8537144298837803
```

1(c) [5 marks] To include a multilayer perceptron with 3 hidden layers in a single node, the classifier was created using

```
clf = MLPClassifier(hidden_layer_sizes=(3,), activation='tanh',
                    max_iter=2000, random_state=0)
```

The histogram of the MLP output (as given by predict_proba) is shown below:



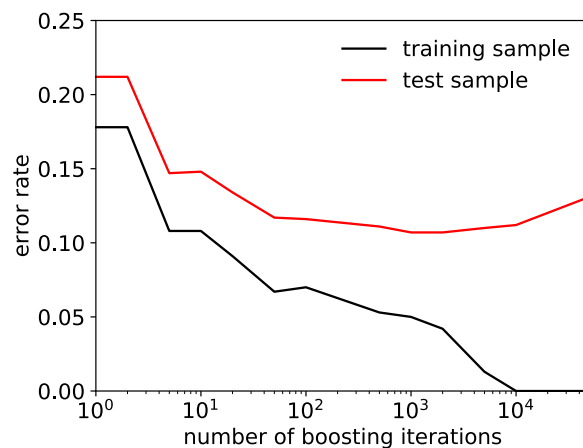
1(d) [5 marks] To select events with the MLP, the function predict_proba is used instead of the decision function. To find the purity, use the efficiencies in Bayes' theorem:

$$\begin{aligned} \text{Signal efficiency} &= \varepsilon_s = P(t > t_c | s) \\ \text{Background efficiency} &= \varepsilon_b = P(t > t_c | b) \\ \text{Signal purity} &= P(s | t > t_c) = \varepsilon_s \pi_s / (\varepsilon_s \pi_s + \varepsilon_b \pi_b) \end{aligned}$$

By defining the critical region with a minimum threshold on its value of 0.5, the program produced the following output:

```
power of test with respect to signal      = 0.9078683259735046
size of test of background                = 0.08270227182144285
purity of signal sample                   = 0.9165104718376038
```

Exercise 2 [bonus]: The total error rate of a BDT (using AdaBoost) as a function of the number of boosting iterations is shown below. The code is given in the appendix. For this exercise, only 1000 events of each type were used. As can be seen in the figure, the error rate from the training sample goes to zero after around 10^4 iterations. For the test sample, a minimum error rate of 10.7% is found at 2000 iterations, with a shallow minimum around 11% from 10^2 to 10^4 iterations. For more than 10^4 iterations the error rate increases slowly.



Appendix – Code listing

```
# simpleClassifier_solution.py
# G. Cowan / RHUL Physics / November 2020
# Simple program to illustrate classification with scikit-learn

import scipy as sp
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

# read the data in from files,
# assign target values 1 for signal, 0 for background
sigData = np.loadtxt('signal1k.txt')
nSig = sigData.shape[0]
sigTargets = np.ones(nSig)

bkgData = np.loadtxt('background1k.txt')
nBkg = bkgData.shape[0]
bkgTargets = np.zeros(nBkg)

# concatenate arrays into data X and targets y
X = np.concatenate((sigData,bkgData),0)
y = np.concatenate((sigTargets, bkgTargets))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
random_state=1)

# Create classifier object and train
# Add code here to include other classifiers (MLP, BDT,...)
# Select classifier
classifier = "LDA"
if classifier == "LDA":
    clf = LDA()
elif classifier == "MLP":
    clf = MLPClassifier(hidden_layer_sizes=(3,), activation='tanh',
max_iter=2000, random_state=0)
elif classifier == "BDT":
    clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
algorithm="SAMME", n_estimators=100)
clf.fit(X_train, y_train)

# Evaluate accuracy using the test data.
# Adjust threshold value tCut or pMin as appropriate

X_bkg_test = X_test[y_test==0]
X_sig_test = X_test[y_test==1]
y_bkg_test = y_test[y_test==0]
y_sig_test = y_test[y_test==1]

if hasattr(clf, "decision_function"):
```

```

    tCut = 0.
    y_bkg_pred = (clf.decision_function(X_bkg_test) >= tCut).astype(bool)
    y_sig_pred = (clf.decision_function(X_sig_test) >= tCut).astype(bool)
else:
    pMin = 0.5
    y_bkg_pred = (clf.predict_proba(X_bkg_test)[: ,1] >=
pMin).astype(bool)
    y_sig_pred = (clf.predict_proba(X_sig_test)[: ,1] >=
pMin).astype(bool)

power = metrics.accuracy_score(y_sig_test, y_sig_pred)
print('power of test with respect to signal = ', power)

# Ex. 1(a) Add code here to obtain the background efficiency
# = size of test alpha = Prob(t >= tCut|bkg)
size = 1. - metrics.accuracy_score(y_bkg_test, y_bkg_pred)
print('size of test of background = ', size)

# Ex. 1(b) Purity of signal sample selected with t >= tCut
piSig = 0.5      # prior probability to be signal
piBkg = 0.5      # prior probability to be background
effSig = power
effBkg = size
purity = effSig*piSig / (effSig*piSig + effBkg*piBkg)
print('purity of signal sample = ', purity)

# Ex. 1(c) make histogram of decision function
plt.figure() # new window
matplotlib.rcParams.update({'font.size':14}) # set all font sizes
tTest = clf.predict_proba(X_test)[: ,1]
if hasattr(clf, "decision_function"):
    tTest = clf.decision_function(X_test) # if available use
decision_function
else:
    tTest = clf.predict_proba(X_test)[: ,1] # for e.g. MLP need to
use predict_proba
tBkg = tTest[y_test==0]
tSig = tTest[y_test==1]
nBins = 50
tMin = np.floor(np.min(tTest))
tMax = np.ceil(np.max(tTest))
bins = np.linspace(tMin, tMax, nBins+1)
plt.xlabel('decision function $t$', labelpad=3)
plt.ylabel('$f(t)$', labelpad=3)
n, bins, patches = plt.hist(tSig, bins=bins, density=True,
histtype='step', fill=False, color='dodgerblue')
n, bins, patches = plt.hist(tBkg, bins=bins, density=True,
histtype='step', fill=False, color='red', alpha=0.5)
plt.savefig("decision_function_hist.pdf", format='pdf')

# Ex. 2 -- total error rate for BDT as function of boosting iterations
boostingIterations = [1, 2, 5, 10, 20, 50, 100, 500, 1000, 2000, 5000,
10000, 50000]
trainErrorRate = []
testErrorRate = []
for i in range(len(boostingIterations)):
    clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
algorithm="SAMME", n_estimators=boostingIterations[i])
    clf.fit(X_train, y_train)

```

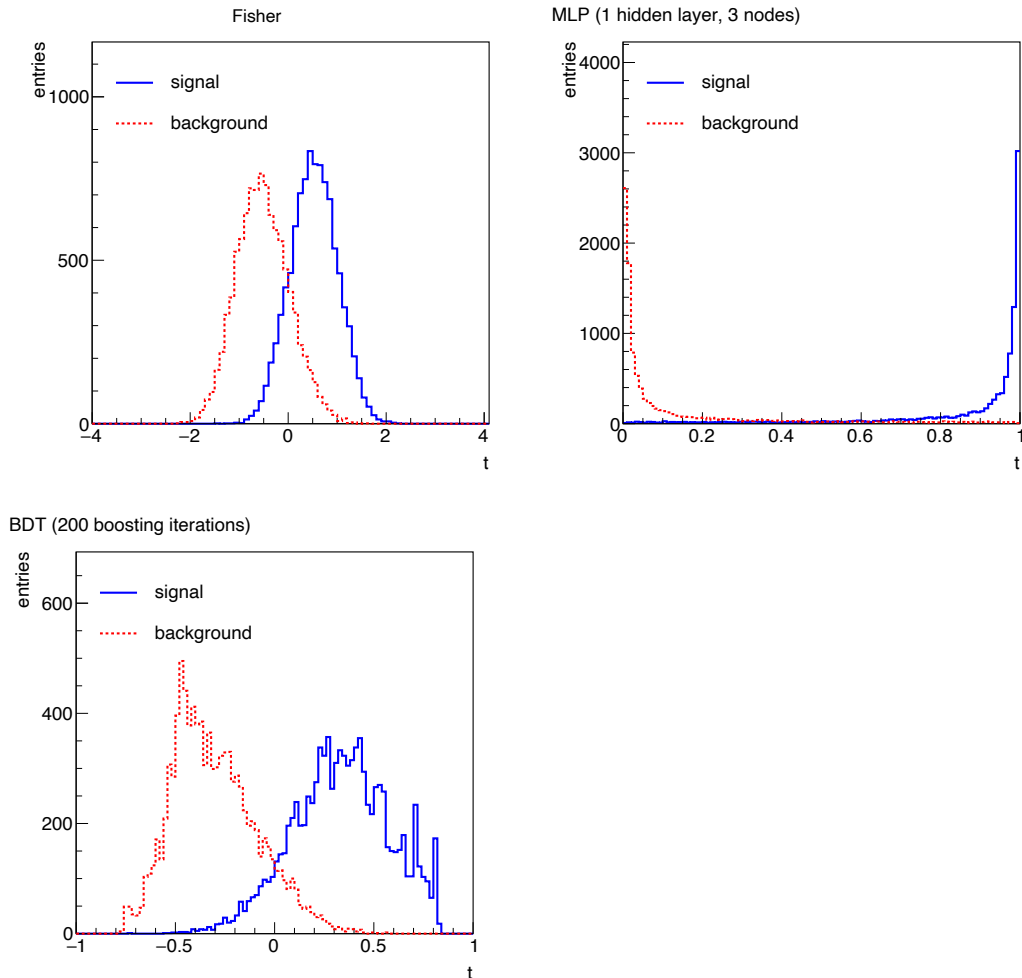
```

    y_pred_train = clf.predict(X_train)
    y_pred_test = clf.predict(X_test)
    trainErrorRate.append(1. - metrics.accuracy_score(y_train,
y_pred_train))
    testErrorRate.append(1. - metrics.accuracy_score(y_test, y_pred_test))
    print(boostingIterations[i], trainErrorRate[i], testErrorRate[i])

# make plot
fig, ax = plt.subplots(1,1)
plt.xlabel(r'number of boosting iterations', labelpad=3)
plt.ylabel(r'error rate', labelpad=3)
plt.xscale('log')
xMin=1.
xMax=5.e4
yMin=0.
yMax=0.25
plt.xlim(xMin, xMax)
plt.ylim(yMin, yMax)
plt.plot(boostingIterations, trainErrorRate, color='black',
label=r'training sample')
plt.plot(boostingIterations, testErrorRate, color='red', label=r'test
sample')
plt.subplots_adjust(left=0.2, right=0.9, top=0.9, bottom=0.2)
handles, labels = ax.get_legend_handles_labels()
plt.legend(handles, labels, loc='upper right', frameon=False)
plt.savefig("BDTErrrorRate.pdf", format='pdf')
plt.show()

```

Using C++/Root, the following distributions were obtained for the Fisher discriminant, MLP (3 nodes in 1 hidden layer) and BDT with 200 boosting iterations:



the following efficiencies and purities were obtained:

signal efficiency (power) Fisher	= 0.8547
background efficiency (size) Fisher	= 0.1719
signal purity (Fisher)	= 0.832554
signal efficiency (power) MLP3	= 0.9134
background efficiency (size) MLP3	= 0.0927
signal purity (MLP3)	= 0.907862
signal efficiency (power) BDT200	= 0.9123
background efficiency (size) BDT200	= 0.0951
signal purity (BDT200)	= 0.905599

The routine `tmvaTrain.cc` was modified to include a multilayer perceptron using

```
factory->BookMethod(dataLoader, TMVA::Types::kMLP, "MLP3",
"H:!V:NeuronType=tanh:HiddenLayers=3");
```


And a boosted decision tree with 200 iterations was created with

```
factory->BookMethod(dataLoader, TMVA::Types::kBDT, "BDT200",  
"NTrees=200:BoostType=AdaBoost");
```

The routine analyzeData.cc was modified to calculate the signal and background efficiencies and signal purity using the code below:

```
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <TFile.h>  
#include <TTree.h>  
#include <TH1D.h>  
#include <TCanvas.h>  
#include <TApplication.h>  
#include <TR00T.h>  
#include <TStyle.h>  
#include <TPad.h>  
#include <TAxis.h>  
#include <TH2F.h>  
#include <TF2.h>  
#include <TLine.h>  
#include "Event.h"  
#include "OutFunc.h"  
#include <TMVA/Reader.h>  
  
using namespace std;  
  
OutFunc* testFisher;      // need global for contour later  
OutFunc* testMLP3;       // need global for contour later  
OutFunc* testBDT200;     // need global for contour later  
  
int main() {  
  
    // Set up an output file and book some histograms  
  
    TFile* histFile = new TFile("analysis.root", "RECREATE");  
    TH1D* hSigFisher = new TH1D("hSigFisher", "Fisher, sig", 100, -5.0, 5.0);  
    TH1D* hBkgFisher = new TH1D("hBkgFisher", "Fisher, bkg", 100, -5.0, 5.0);  
    TList* hList = new TList();      // list of histograms to store  
    hList->Add(hSigFisher);  
    hList->Add(hBkgFisher);  
  
    TH1D* hSigMLP3 = new TH1D("hSigMLP3", "MLP3, sig", 100, -5.0, 5.0);  
    TH1D* hBkgMLP3 = new TH1D("hBkgMLP3", "MLP3, bkg", 100, -5.0, 5.0);  
    hList->Add(hSigMLP3);  
    hList->Add(hBkgMLP3);  
  
    TH1D* hSigBDT200 = new TH1D("hSigBDT200", "BDT200, sig", 100, -5.0, 5.0);  
    TH1D* hBkgBDT200 = new TH1D("hBkgBDT200", "BDT200, bkg", 100, -5.0, 5.0);  
    hList->Add(hSigBDT200);  
    hList->Add(hBkgBDT200);  
  
    // Set up the OutFunc object.  First argument must be one of the classifiers.  
    // 4th argument is offset for making contour; otherwise should be zero.  
    // 5th argument is bool array indicating which variables were used in training  
  
    std::string dir = "../train/dataset/weights/";  
    std::string prefix = "tmvaTest";  
    const double tCutFisher = 0.;  
    const double tCutMLP3 = 0.5;  
    const double tCutBDT200 = 0.;  
    std::vector<bool> useVar(3);  
    useVar[0] = true;      // x  
    useVar[1] = true;      // y  
    useVar[2] = true;      // z  
    testFisher = new OutFunc("Fisher", dir, prefix, 0., useVar);  
    testMLP3 = new OutFunc("MLP3", dir, prefix, 0., useVar);
```

```

    testBDT200 = new OutFunc("BDT200", dir, prefix, 0., useVar);

// Open input file, get the TTrees, put into a vector

TFile* inputFile = new TFile("../generate/testData.root");
inputFile->ls();
TTree* sig = dynamic_cast<TTree*>(inputFile->Get("sig"));
TTree* bkg = dynamic_cast<TTree*>(inputFile->Get("bkg"));
std::vector<TTree*> treeVec;
treeVec.push_back(sig);
treeVec.push_back(bkg);

// Loop over TTrees, i=0 is signal, i=1 is background

int nSig, nBkg;
int nSigAccFisher = 0;
int nSigAccMLP3 = 0;
int nSigAccBDT200 = 0;
int nBkgAccFisher = 0;
int nBkgAccMLP3 = 0;
int nBkgAccBDT200 = 0;
for (int i=0; i<treeVec.size(); i++){

    treeVec[i]->Print();
    Event evt;
    treeVec[i]->SetBranchAddresses("evt", &evt);
    int numEntries = treeVec[i]->GetEntries();
    if ( i == 0 ) { nSig = numEntries; }
    if ( i == 1 ) { nBkg = numEntries; }
    std::cout << "number of entries = " << numEntries << std::endl;

// Loop over events.

    for (int j=0; j<numEntries; j++){

        treeVec[i]->GetEntry(j); // sets evt
        double tFisher = testFisher->val(&evt); // evaluate test statistic
        double tMLP3 = testMLP3->val(&evt);
        double tBDT200 = testBDT200->val(&evt);

        if ( i == 0 ){ // signal
            hSigFisher->Fill(tFisher);
            hSigMLP3->Fill(tMLP3);
            hSigBDT200->Fill(tBDT200);
            if ( tFisher >= tCutFisher ) { nSigAccFisher++; }
            if ( tMLP3 >= tCutMLP3 ) { nSigAccMLP3++; }
            if ( tBDT200 >= tCutBDT200 ) { nSigAccBDT200++; }
        }
        else if ( i == 1 ){ // background
            hBkgFisher->Fill(tFisher);
            hBkgMLP3->Fill(tMLP3);
            hBkgBDT200->Fill(tBDT200);
            if ( tFisher >= tCutFisher ) { nBkgAccFisher++; }
            if ( tMLP3 >= tCutMLP3 ) { nBkgAccMLP3++; }
            if ( tBDT200 >= tCutBDT200 ) { nBkgAccBDT200++; }
        }
    }
}

// Compute efficiencies (power, size) etc.

double epsSigFisher = static_cast<double>(nSigAccFisher)
    / static_cast<double>(nSig);
std::cout << "signal efficiency (power) Fisher = " << epsSigFisher
    << std::endl;
double epsBkgFisher = static_cast<double>(nBkgAccFisher)
    / static_cast<double>(nBkg);
std::cout << "background efficiency (size) Fisher = " << epsBkgFisher
    << std::endl;

double epsSigMLP3 = static_cast<double>(nSigAccMLP3)
    / static_cast<double>(nSig);

```

```

std::cout << "signal efficiency (power) MLP3 = " << epsSigMLP3
    << std::endl;
double epsBkgMLP3 = static_cast<double>(nBkgAccMLP3)
    / static_cast<double>(nBkg);
std::cout << "background efficiency (size) MLP3 = " << epsBkgMLP3
    << std::endl;

double epsSigBDT200 = static_cast<double>(nSigAccBDT200)
    / static_cast<double>(nSig);
std::cout << "signal efficiency (power) BDT200 = " << epsSigBDT200
    << std::endl;
double epsBkgBDT200 = static_cast<double>(nBkgAccBDT200)
    / static_cast<double>(nBkg);
std::cout << "background efficiency (size) BDT200 = " << epsBkgBDT200
    << std::endl;

// Compute signal purity assuming equal prior probabilities
double piSig = 0.5;
double piBkg = 0.5;

double purityFisher = epsSigFisher*piSig /
    (epsSigFisher*piSig + epsBkgFisher*piBkg);
std::cout << "signal purity (Fisher) = " << purityFisher << std::endl;

double purityMLP3 = epsSigMLP3*piSig/(epsSigMLP3*piSig + epsBkgMLP3*piBkg);
std::cout << "signal purity (MLP3) = " << purityMLP3 << std::endl;

double purityBDT200 = epsSigBDT200*piSig /
    (epsSigBDT200*piSig + epsBkgBDT200*piBkg);
std::cout << "signal purity (BDT200) = " << purityBDT200 << std::endl;

// Close up
inputFile->Close();
histFile->Write();
histFile->Close();

return 0;
}

```

Total error rate versus number of boosting iterations of BDT (using 1000 training events):

