DIANA Fellowship

Improving the Boosted Decision Trees in TMVA

Andrew Carnes (University of Florida), Sergei Gleyzer (University of Florida)

January 25, 2017

Abstract

This paper presents the work completed for the 2016 DIANA fellowship, which took place in October, November, and December of that year. As part of the DIANA fellowship, the Boosted Decision Trees (BDTs) in TMVA were improved on three fronts. First, the loss function capability was abstracted and the choice of metrics was expanded from a single loss function to three. Second, the training was parallelized via multi-threading. Lastly, the time needed to predict a set of events was reduced by debugging the evaluation algorithm. In addition, the TMVA Users Guide was updated, documenting the appropriate information, and a Jupyter notebook illustrating the new loss functions has been made available online.

Contents

1 Introduction

As a requirement of the DIANA fellowship, this paper decribes in detail the work done for organization throughout the three month period of October, November, and December in 2016. During this time period, various improvements were made to the Toolkit for Multivariate Analysis (TMVA), a package in ROOT, by enhancing the capabilities of the Boosted Decision Trees (BDTs) and improving the speed of the algorithm. The hard-coded loss function was replaced with multiple options, broadening the utility of the algorithm, the training was parallelized, reducing the time needed to train the trees, and the evaluation algorithm was revised, reducing the time needed to predict a set of events. A Jupyter notebook showing the new loss function capabilities was uploaded to SWAN, providing the first notebook for regression. In addition, the TMVA Users Guide has been edited to document the new features.

In order to outline these enhancements in detail, the general BDT algorithm will first be described. After covering the general procedure, TMVA's specific implementation will be covered. Then, after understanding the baseline TMVA BDT algorithm, the improvements to the BDTs will be detailed. Plots illustrating the new capabilities and the improvements in speed in this new version of the BDTs will also be presented.

2 Boosted Decision Trees

The BDT algorithm is a supervised machine learning algorithm: it learns from a set of data with known answers in order to predict data with unknown answers. The algorithm creates a forest, which is a collection of trees. The forest starts with a single tree, which chops up the feature space into discrete regions, fitting constants in each as to minimize the total error between between the true values and the predicted values of the tree. The function defining the net error is called the loss function. If the predictions by the tree are general enough the events will be off by some amount, i.e. the predicted values will still differ from the true values. Since the events are still off, the algorithm will add another tree attempting to correct the predictions of the last tree to further reduce the error. The algorithm will continue to add trees further correcting the predictions and minimizing the error until it reaches the maximum number of trees, a hyperparmeter set by the user.

2.1 A Single Decision Tree

The decision tree needs to decide how to partition the space such that it is computationally efficient and reduces the error defined by the loss function, $L(z, \hat{z})$, where z is the vector of true values and $\hat{z}$ is the vector of predicted values. To meet these demands the tree building algorithm takes a greedy approach, recursively dividng the regions into two until the maximum number of terminal nodes, a hyperparameter determined by the user, is reached. The maximum number of terminal nodes corresponds to the number of predictive regions in feature space. The algorithm goes as follows. First fit the entire feature space with a constant. Then search along each feature proposing a cut in the form of a hyperplane through a value of that feature, which would divide the space/subspace into two new regions. The two new

proposed regions are fit with constants – all of the events falling in the same region are fit with the same value – where each time the constant chosen for a region, R, is the value that minimizes the loss function there.

$$\frac{d}{dc_R} L(z, \hat{z} = c_R) = 0 | z, \hat{z} \in R \tag{1}$$

The loss function before the subdivision, $L_{before} = L(z, \hat{z}_{before})$, is compared to the loss function after the subdivision, $L_{after} = L(z, \hat{z}_{after})$, and the error reduction is calculated like so, $-(L_{after} - L_{before})$, which is a positive number since L_{after} has a smaller error than L_{before}. Call R_i the region to split and $R_{i,1}$ and $R_{i,2}$ the resulting regions after the proposed split.

$$\text{Error Reduction} = \sum_{j \in R_i} [z - c_i]^2 - \sum_{j \in R_{i,1}} [z - c_{i,1}]^2 - \sum_{j \in R_{i,2}} [z - c_{i,2}]^2 \tag{2}$$

The cut with the largest error reduction of those searched is the best cut for the feature. The algorithm checks the other features in the same manner, reporting the best cut and error reduction for each. The cut chosen for the region is the cut with the largest error reduction of all the features. Now the feature space is divided into two regions, each with constant fits. The algorithm then repeats the same process in each subregion, figuring out the best cut and constants for each, eventually splitting the region that provides the maximum gain in error reduction. This process continues, each time splitting the subspace that yields the best return, until the number of regions equals the maximum number of terminal nodes set by the user.

2.2 Building a Single Decision Tree: an Example

Here is an example of the tree building algorithm using Least Squares as the loss function. Consider a case with N events where each event has two features x and y, and each event is uniquely labeled by the subscript i. The maximum number of terminal nodes in this example is 3.

$$\text{Least Squares} \equiv \frac{1}{2} \sum_{i=1}^{N} [z(x_i, y_i) - \hat{z}(x_i, y_i)]^2 \tag{3}$$

$$\frac{d}{dc_R} \frac{1}{2} \sum_{i=1}^{N} [z(x_i, y_i) - c_R]^2 = 0 \rightarrow c_R = \frac{1}{N} \sum_{i=1}^{N} z_i \tag{4}$$

Take a look at the the top left of Figure 1, the events for the example were generated such that those with x>0.5 have z=10 those with x<0.5 and y>0.5 have z=-8, and those with x<0.5 and y<0.5 have z=-12. The colored regions represent the true values of the underlying true distribution and the crosses represent the generated events themselves. In the end the tree with three terminal nodes should model these generated regions, correctly predicting the events. In the next three pictures in Figure 1 the colors of the crosses still represent the true values, but the colored regions represent the fits in the decision tree regions at each iteration.

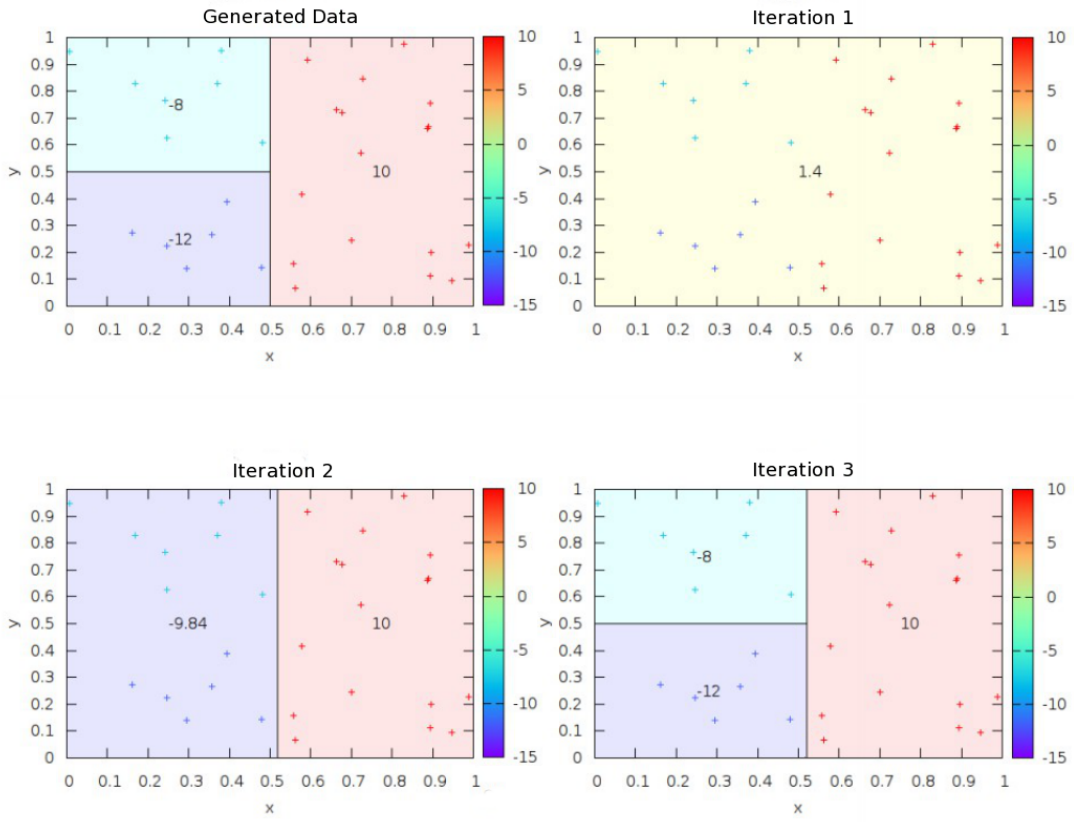

Figure 1: The stages of creation for a single decision tree with three terminal nodes.

Equation 3 determines that the constant fit in a region should be the mean of the true values in the region. So in iteration 1 the tree fits all of the events with the mean, 1.4. Then in iteration 2 the tree searches along x and y calculating the error reduction for each possible cut along each feature. The cut with the maximum error reduction turns out to be x=0.535. The left region is fit with its mean, -9.84 and the right region with its mean, 10. The tree then searches in both subspaces along x and y for the best split in each space, choosing to split the region with the most gain. The algorithm finds the maximum error reduction in the left region with a cut at y=0.5, fitting the top with -8 and the bottom with -12. At the end of iteration 3 there are three terminal nodes and the algorithm stops building the tree. The tree has correctly modeled the underlying distribution of the true values.

Notice that the fit for an event is a given by a series of binary decisions, and that the entire model is given by a tree of binary decisions, hence the name. The tree structure is illustrated for the previous example in Figure 2.

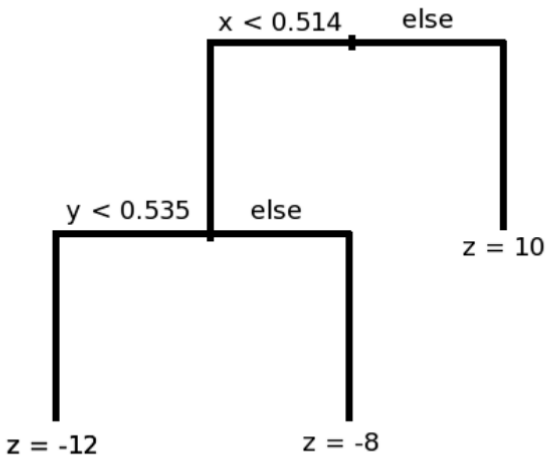

Figure 2: The decision tree model gets its name from the fact that it can be viewed as a tree of binary decisions.

2.3 The Decision Tree Algorithm

Now that the basic concepts behind the decision tree have been covered, the decision tree algorithm is outlined in pseudocode.

```
node                  |  tree
  bestSplitValue      |    rootNode
  bestSplitFeature    |    terminalNodes
  bestErrorReduction  |    trainingEvents
  fitValue            |    loss_function
                      |    nodeLimit
  mother              |
  leftDaughter        |
  rightDaughter       |
                      |
  loss_function       |

node::fitAndCalcOptimumSplit(loss_function)
# General outlining the necessary steps to calculate
# the best split value, feature, and error reduction
# for the node

  bestSplitValue = 0
  bestSplitFeature = -1
  bestErrorReduction = -1
```

```
  # get the constant fit value in this region aka subspace aka node
  # based upon the constant that minimizes the loss function
  fitValue = loss_function.fit_constant(events)

  # calculate the error reduction for each feature
  # and for a set of points along that feature
  # store the information for the best one
  for(f in features)
      for(splitPoint in f)
          errorReduction = loss_function.calculateErrorReduction(events, splitPoint, f)
          if(errorReduction > bestErrorReduction)
            bestErrorReduction = errorReduction
            bestSplitFeature = f
            bestSplitValue = splitPoint

tree::buildTree()
# build a tree up to the given node limit

  # set up the root node for the tree
  rootNode = new node
  rootNode.events = trainingEvents
  terminalNodes.add(rootNode)

  # keep track of the node with the best error reduction
  # during the loop through the nodes
  bestNodeErrorReduction = -1
  nodeToSplit = 0

  # set fit value and figure out the best split
  # feature, split value, and best error reduction
  # for the root node
  if(terminalNodes.size() == 1)
    rootNode.fitAndCalcOptimumSplit(loss_function)

  # All terminal nodes have best split info available
  # see which one is the best one to split
  for(node in terminalNodes)
    if(node.bestErrorReduction < bestNodeErrorReduction)
      bestNodeErrorReduction = node.bestErrorReduction
      nodeToSplit = node

  # found best terminal node aka subspace to split
  # link mother and daughter nodes
  left = new node
  right = new node
```

```
nodeToSplit.leftDaughter = left
nodeToSplit.rightDaughter = right
left.mother = nodeToSplit
right.mother = nodeToSplit

# filter the events appropriately from the mother space into the subspaces
for(event in nodeToSplit.events)
  if(event.feature[nodeToSplit.bestSplitFeature] < nodeToSplit.bestSplitValue)
    left.events.append(event)
  else
    right.events.append(event)

# calculate the best split info for the new nodes
# also figure out the constant fits for each region
left.fitAndCalcOptimumSplit(loss_function)
right.fitAndCalcOptimumSplit(loss_function)

# nodeToSplit has been divided into subspaces
# it is no longer a terminal node, but the daughters are
terminalNodes.remove(nodeToSplit)
terminalNodes.add(left)
terminalNodes.add(right)

# continue greedily dividing until nodeLimit is reached
if(terminalNodes.size() < nodeLimit) buildTree()
```

2.4 Boosting

Boosting involves iteratively adding trees to create a forest. Each subseuqent tree corrects the predictions of the earlier trees.

$$\text{Forest}(\vec{x}) = T_0(\vec{x}) + T_1(\vec{x}) + ... + T_n(\vec{x}) \tag{5}$$

Now T_0 is the initial fit and fits the true values as well as possible. Then, T_1 is created such that it corrects the predictions of T_0. After, T_2 corrects $T_0 + T_1$, and in general the n^{th} tree, T_n, corrects the predictions given by the previous n trees. This is analogous to perturbation theory in physics. The values the n^{th} tree should model at each iteration are given by $d\hat{z}_n$ such that this correction heads in the direction of the minimum at that stage. Therefore $d\hat{z}_n$ should be in the same direction as the negative of the gradient. And the nth corrective term to the forest, T_n, should model $d\hat{z}_n$ up to some constant.

$$-\frac{d}{dT_n(x^i)} L(z(x^i), T_0(x^i) + ... + T_{n-1}(x^i)) \rightarrow d\hat{z}_n^i \tag{6}$$

In this way the n^{th} tree would target the $d\hat{z}_n$ values for each event i given by the equation

above and the tree would choose the appropriate splits and constants as to minimize the loss function, with the n^{th} targets acting as the true values for the tree. However, computationally efficient methods exist for building a tree with Least Squares, so in practice the tree will fit the n^{th} targets for the specific loss function using Least Squares, rather than the loss function itself. This guarantees that the algorithm heads downhill, but further steps can be taken to reach a more minimum value. After the tree has modeled the gradient by forming the regions in feature space, and fitting them with constants, the constants in those regions can be recalculated as to best minimize the loss function within each region.

$$\frac{d}{dc_{n,R}} L(z, \hat{z} = T_0 + T_1 + ... + T_{n-1} + c_{n,R}) = 0 \rightarrow c_{n,R} \tag{7}$$

If the user wishes to head downhill slower or quicker the user can supply a learning rate value, which scales each tree's predictions by a constant. If the learning rate were 0.3 then the constant fits in each of the terminal regions would be scaled by 0.3 for every tree. Trees are added until the number of trees in the forest reaches the maximum number of trees specified by the user as a hyperparameter.

2.5 The BDT Algorithm

Now that the concepts behind the decision tree and boosting have been covered, the BDT algorithm is outlined in pseudocode.

```
forest
  trainingEvents
  trees
  loss_function

  nodeLimit
  treeLimit
  learningRate

forest::buildForest()

  while(trees.size() < treeLimit)
    # head downhill by setting the targets for the tree
    # the target is the gradient for the loss function
    # covered earlier, each tree fits the targets as if
    # they were the true values
    for(event in trainingEvents)
      loss_function.set_target(event)

    # build the tree
    # the tree usually uses least_squares to fit the targets
    # since it's efficient
```

```
tree = new tree
trees.add(tree)
tree.buildTree(nodeLimit)

# Now recalculate the best fits in the terminal regions
for(tnode in tree.terminalNodes)

  # constant that minimizes the loss function in the region
  tnode.fitValue = loss_function.fit_constant(tnode.events)

  # scale by learningRate, hyperparameter set by user
  tnode.fitValue*=learningRate

  # update the predictedValue now that the event has
  # been fit by a new tree
  for(event in tnode.events)
      event.predictedValue+=tnode.fitValue
```

3 The TMVA BDT Implementation

This section outlines TMVA's implemention of the BDT algorithm for regression. The base implementation is covered here so that the improvements for the DIANA fellowship may be laid out in detail in a later section of the paper.

3.1 TMVA Decision Tree Algorithm

As aforementioned regression tasks normally use Least Squares to fit the targets for the tree. This is indeed the case for TMVA. As such, it is important to cover some background information about the Least Squares error reduction calculation in order to understand why certain quantities are calculated in the algorithm. After covering the background information TMVA's decision tree algorithm is outlined in pseudocode.

3.1.1 Background Information

For regression tasks, TMVA uses Least Squares to fit the targets and build each tree. For Least Squares the loss in a region is given by the following equation, where z is the target for an event in the region and $< z >$ is the mean of the targets in the region.

$$Loss = \sum_{i \in R} (z_i - < z >)^2 \tag{8}$$

Which after simplifying yields the following equation for the loss in a given region, where N is the number of events in the region.

$$Loss = N(< z^2 > - < z >^2) \tag{9}$$

Simplifying even further yields an even better equation.

$$Loss = N(\frac{1}{N}\sum_{i\in R} z_i^2 - \frac{1}{N}\sum_{i\in R} z_i \frac{1}{N}\sum_{i\in R} z_i) \tag{10}$$

$$Loss = \sum_{i\in R} z_i^2 - \frac{1}{N}\sum_{i\in R} z_i \sum_{i\in R} z_i \tag{11}$$

Subtracting the loss after the split from the loss before the split yields the error reduction. Before the split there is a single region. After the split the region is divided into two subregions with their own constant fits. Values of the feature less than the cut on the feature make up the left region and values of the feature greater than the cut make up the right region. See Figure 1 for an example.

$$ER = Loss_R - Loss_{R,left} - Loss_{R,right} \tag{12}$$

To calculate the error reduction the aglorithm must calculate the sum of the targets and the sum of the squared targets in the mother region and the proposed daughter regions.

3.1.2 The DT Algorithm

TMVA's decision tree algorithm goes as follows, utilizing two functions: BuildTree(eventSample, node) and TrainNodeFast(eventSample, node).

```
BuildTree(eventSample, node)

  # base case to stop building the tree
  if(node.depth >= maxDepth) return

  node.target = 0
  node.target2 = 0
  node.nevents = eventSample.size()

  # define this part as build-sum
  # Calculate the sum of targets and sum of targets squared
  # Figure out xmin and xmax for each feature
  for(event in eventSample)
    node.target+=event.target
    node.target2+=event.target*event.target

    for(var in features)
      if(event.feature(var) > xmax[var]) xmax[var] = event.feature(var)
      if(event.feature(var) < xmin[var]) xmin[var] = event.feature(var)

  # define this process as build-train
  # calculate optimum split and error reduction
```

```
TrainNodeFast(eventSample, node)

# link the nodes together
right = new node
left = new node
link_nodes(left, right, mother=node)

# prepare event samples to continue building the tree
rightSample
leftSample

# define this process as build-filter
# filter events into appropriate daughter regions
for(event in eventSample)
    if(event.feature[node.bestSplitFeature] < node.bestSplitCut) leftSample.add(event)
    else rightSample.add(event)
```

3.1.3 DT Algorithm Overview

The major work is done by the following processes labeled in the above pseudocode. These will be referenced later when discussing the multi-threading improvements.

build-sum Loops over the events in the node to calculate the sum of targets, squared targets, and to find the min and max for each feature. Since this loops over the number of events this is a major contribution to the time for the algorithm and is targeted for parallelization.

build-train Calculates the optimum split criteria for the node and the net error reduction. This also loops over the events and is a target for parallelization. This is the major part of the tree building algorithm and is described in greater detail in the next section.

build-filter Loops over the events in the node to filter them to the left and right daughter nodes. Again this loops over the events in the node and is a major contribution to the time of the algorithm. This involves appending to the end of a list which can't be done efficiently in parallel. Two processes acting at the same time will see the same end iterator causing problems.

3.1.4 build-train: TrainNodeFast

The build-train portion of TMVA's decision tree algorithm is the major piece and will be described in detail here. This portion is implemented by TrainNodeFast. TrainNodeFast creates a histogram of the number of events, as well as the target and squared target values for every feature. The histograms will be refered to as N, target, and target2, respectively. For example, target[f][b] would provide the sum of the target values for all events whose feature value falls into the range for that bin, and target[f] would be the entire histogram

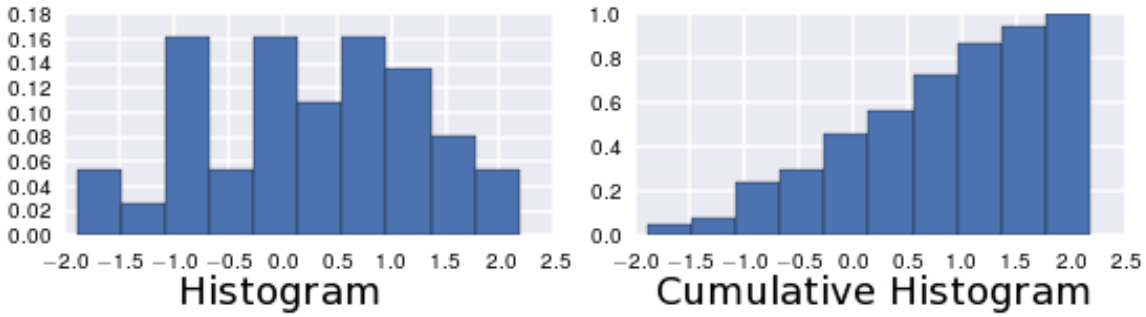

Figure 3: A histogram and cumulative histogram for some feature. The second bin (starting to count at 1) corresponds to a split point of about -1.0. And the second bin in the cumulative histogram gives the sum of the first and second bin of the histogram on the left. The cumulative histogram is a convenient container for the sums needed in the error reduction calculation covered earlier.

for that feature. The histogram target2 is analogous, and N works the same way except that each bin is the number of events in that range.

Then TrainNodeFast goes on to create cumulative sum histograms for all three. The two dimensional arrays are edited in place, such that N, target, and target2 no longer represent their histograms but cumulative sum histograms. With the cumulative sums in hand, target[feature][bin] will contain the sum of the target values with a feature value less than or equal to the split point corresonding to that bin. So target[f][bin] is the sum of the target values in the proposed left daughter. Similarly target2[feature][bin] will contain the sum of the squared targets for the left daughter. And N[feature][bin] will contain the number of events in the left daughter.

Each bin then serves as a proposed split point, and the three cumulative histograms contain the information needed to calculate the error reduction for each split. Now it's simply a matter of looping over the features and the bins in the histograms to calculate the error reduction for each possible point, keeping track of the information for the maximum.

3.1.5 The TrainNodeFast Algorithm

Having covered the tricky part of the algorithm, the pseudocode is now presented. There's a lot of initialization, but some of the initialization code was parallelized for the case that where NBINS is close to the order of the amount of training data, so it's included here for reference.

```
TrainNodeFast(eventSample, node)

  # the best split info for the node
  bestErrorReduction = -1
  bestSplitFeature = -1
  bestSplitValue = -1

  # the best split info for each feature
```

```
# initialize all values to -1
bestSplitValueArray     = array(nfeatures, -1)
bestErrorReductionArray = array(nfeatures, -1)

# the number of bins for the histograms along each feature
nbins   = array(nfeatures)

# a target, target2, and N histogram for each feature
target  = array(nfeatures)
target2 = array(nfeatures)
N       = array(nfeatures)

# build-train-init-hists
# initialize the histogram information
# depending on whether the feature is continuous or it is an integer
for(f in features)
  # integers get one bin for every integer in the range
  if(f is an integer)
    nbins[f]   = xmax[f] - xmin[f] +1
    target[f]  = array(nbins[f])
    target2[f] = array(nbins[f])
    N[f]       = array(nbins[f])
  # continuous features get NBINS set by the user as a hyperparameter
  else
    target[f]  = array(NBINS)
    target2[f] = array(NBINS)
    N[f]       = array(NBINS)

 # initialize all the histogram bins to zero
 for(bin in range(0,nbins[f]))
   N[f][bin]       = 0
   target[f][bin]  = 0
   target2[f][bin] = 0

# continue to initialize histogram information
for(f in features)
    # calculate these to use later when figuring out
    # which bin an event belongs in
    if(f is an integer)
      invBinWidth[f] = 1
      binWidth[f] = 1
    else
      binWidth[f] = (xmax[f] - xmin[f]) / nbins
      invBinWidth[f] = 1/binWidth[f]
```

```
    # map each split value for each feature to each bin
    for bin in nbins
      cutValues[f][bin] = xmin[f] + (bin+1.0)*binWidth[f]

# done with build-train-init-hists

# build-train-fill-hists
# now fill the histograms by looping over the events and features
for(event in eventSample)
  nTotal+=1
  for(f in features)
    bin = min(nbins[f]-1, max(0,invBinWidth[f]*(event.feature_value(f)-xmin[ivar])))
    target[f][bin]  += event.target
    target2[f][bin] += event.target*event.target
    N[f][bin]       += 1

# build-train-accumulate-hists
# now turn the histograms into cumulative histograms
for(f in features)
  for(bin in range(1,nbins[f])
    target[f][bin]  += target[f][bin-1]
    target2[f][bin] += target2[f][bin-1]
    N[f][bin]       += N[f][bin-1]

# build-train-calc-best-error
# finally calculate best error reduction for each feature
for(f in features)
  for(bin in range(0,nbins[f]))
    errReduction = getErrorReduction(N[f][bin], target[f][bin], target2[f][bin],
                                     node.nevents, node.target, node.target2)

    # if this is better than the current for the feature save it
    # and the related info
    if(errReduction > bestErrorReductionArray[f])
      bestErrorReductionArray[f] = errReduction
      bestSplitValueArray[f] = cutValues[f][bin]

# the best error reduction for the node is the best of the features
for(f in features)
  if(bestErrorReductionArray[f] > bestErrorReduction)
      bestErrorReduction = bestErrorReductionArray[f]
      bestSplitValue     = bestSplitValueArray[f]
      bestSplitFeature   = f

# save the information into the node
```

```
  node.bestErrorReduction = bestErrorReduction
  node.bestSplitValue = bestSplitValue
  node.bestSplitFeature = f
```

3.1.6 build-train Overview

build-train-init-hists Loops over bins for each feature to initialize histograms. This may take a long time if the number of bins*features is on the order of the amount of training data.

build-train-fill-hists Loops over features for every event to put the event info into the correct bin in each histogram. This is the major part of the TrainNodeFast process and the most important to parallelize.

build-train-accumulate-hists Loops over bins for each feature to turn histograms into cumulative histograms. If bins*features is on the order of the amount of training data then this may take a while as well.

build-train-calc-best-error Loops over the bins for each feature to calculate the best error info. Again this may take a while in the case where bins*features is on the order of the amount of data.

3.2 TMVA Boosting Algorithm

This section covers TMVA's implementation of boosting. The appropriate background information necessary to understand the pesudocode has been introduced, so the algorithm is presented right away. A summary follows after the pseudocode.

```
BuildForest(...)

  # init-events: happens once before building the trees

  # init some parameters for the huber loss function
  # return the constant fit for the sample that minimizes the
  # loss function
  initial_fit_constant = huber_loss_function.init(trainingEvents)

  # set the initial predicted value for all events
  for(event in trainingEvents)
    event.predictedValue = initial_fit_constant

  # boost-update-targets
  # set the targets for the next tree
  # loops over all the events in the sample
  huber_loss_function->SetTargets(trainingEvents)
```

```
# done with init-events

# build-forest-loop
while(trees.size() < treeLimit)
  tree = new tree
  # build-tree
  tree.BuildTree(trainingEvents, tree.rootNode)
  trees.add(tree)

  # done building tree time to do boosting
  # boost-tree

  # boost-leaf
  # loop over events to figure out which terminal node
  # each event falls into d
  tnodeMap = map<nodes,listofevents>()
  for(event in events)
    node = tree.getTerminalNodeForEvent(event)
    tnodeMap[node].add(event)

  # boost-fit
  # now that the events in each terminal node are known
  # the loss function can set the fit in each terminal node
  for(tnode, eventList in map)
    fit_constant = huber_loss_function.Fit(eventList)
    tnode.SetFit(learning_rate*fit_constant)

  # boost-update-predictions
  # now that the events have been fit by a new tree, update their
  # predicted values
  for(event in trainingEvents)
    event.predictedValue += tree.getTerminalNodeForEvent(event).getFit()

  # boost-update-targets
  # set the targets for the next tree
  # loops over all the events in the sample
  huber_loss_function->SetTargets(trainingEvents)

  # done boosting, done with boost-tree
```

3.3 TMVA BDT Algorithm Overview

This section outlines the entire TMVA BDT algorithm, so that the processes can be easily referenced when explaining the improvements.

init-events Happens once before building the set of trees. Intializes some parameters in the loss function. Get an initial constant fit for the events from the loss function. Loop over all events to set the initial predicted values to the fit constant. Then loop over all events to set the targets for the first tree. This contributes a small amount to the total time, since it's not in the forest loop.

build-forest-loop Build trees until the maximum number of trees has been reached.

- **build-tree** Build the tree. This function repeats itself recursively until there are enough nodes in the tree.
  - **build-sum** Loop over the events to get the sum of the targets, targets squared, and to calculate xmin and xmax for each feature for events in the node.
  - **build-train** Calculate the best split feature, split value, and error reduction for the node.
    - **build-train-init-hists** Involves a feature loop and a loop over the bins inside the feature loop to initialize the histograms for each feature.
    - **build-train-fill-hists** For each event loop over the features to add the event information to the appropriate bin in the histograms of each feature.
    - **build-train-accumulate-hists** For each feature loop over the bins to turn the histograms into cumulative histograms.
    - **build-train-calc-best-error** For each feature loop over the bins to calculate the error reduction for each split value. Save the info for the best split.
  - **build-filter** Now that the split information is calculated loop over the events adding those on the left of the cut to the leftSampleList and those on the right of the cut to the rightSampleList. These lists will be passed to BuildTree to continue recursively building the tree until the max depth is reached.
- **boost-leaf** Loop through the events, figure out which terminal node the event belongs in and add it to the list corresponding to that terminal node.
- **boost-fit** Now that the lists of events in each termal node are finished, for each list calculate the fit for the terminal node.
- **boost-update-predictions** The fits in terminal nodes are known, loop through the events to update their predictions based upon each's terminal node.
- **boost-update-targets** The event predictions have been updated. Loop through the events to set the targets for the next tree.

4 Improvements to the TMVA BDTs

4.1 Abstracting the Loss Function Capability

As seen in the pseudocode, TMVA implemented a single hard-coded loss function for the boosting. Different loss functions create different trees depending upon the events the loss function focuses on. Having different options for the loss functions allows the user to focus on the events they care about and predict those more accurately. Relative to Least Squares, Huber focuses on the core of the residuals, those events with small $|z - \hat{z}|$ values, while discounting events with large $|z - \hat{z}|$ values. The user may very well want to focus on the hard to predict events and reign them in.

As part of the fellowship, an abstract LossFunctionBDT class was created and the hard-coded Huber processes were replaced with methods of the abstract class. Three functions were then coded to implement the abstract LossFunctionBDT class: LeastSquaresLossFunctionBDT, AbsoluteDeviationLossFunctionBDT, and HuberLossFunctionBDT. The user can then choose which one to build the forest with.

4.1.1 LossFunctionBDT

LossFunctionBDT is the abstract loss function class. All of the methods require implementation when inherited. The fit is the constant that minimizes the loss function for the events in the node. The target is the target for the upcoming tree in the forest to model. It is the negative gradient of the loss function with respect to the predicted value for an event. Init must initialize the parameters for the loss function if they exist and provide the events with an initial fit.

```
class LossFunctionBDT
  virtual Init(events)                  # initialize the loss function
  virtual SetTargets(events)            # set targets for collection of events
  virtual Target(event)                 # set target for one event
  virtual Fit(events)                   # set fit for collection of events in a node
```

4.1.2 LeastSquaresLossFunctionBDT

This loss function inherits LossFunctionBDT and implements the functions with the appropriate behavior for this error metric.

$$\text{Least Squares} \equiv \frac{1}{2}\sum_{i=1}^{N}[z^i - \hat{z^i}]^2 \tag{13}$$

Taking a derivative with respect to the ith predicted value yields the residual. Setting $\hat{z}$ to a constant for all i and taking a derivative with repect to $\hat{z}$ yields the constant of best fit, the mean of the residuals.

```
class LeastSquaresLossFunctionBDT
  Init(events)  # set initial predictions to the mean
    mean = Fit(events)
```

```
    for(event in events)
      event.predictedValue+=mean

  SetTargets(events)
    for(event in events)
      event.target = Target(event)

  Target(event)  # residual
    return event.trueValue - event.predictedValue

  Fit(events)    # mean
    mean = 0
    for(event in events)
      mean+=event.trueValue-event.predictedValue
    mean=mean/events.size()
    return mean;
```

4.1.3 AbsoluteDeviationLossFunctionBDT

This loss function inherits LossFunctionBDT and implements the functions with the appropriate behavior for this error metric.

$$\text{Absolute Deviation} \equiv \sum_{i=1}^{N} |z^i - \hat{z^i}| \tag{14}$$

Taking a derivative with respect to the ith predicted value yields +1 for $z > \hat{z}$ and -1 for $z < \hat{z}$. Setting $\hat{z}$ to a constant for all i and taking a derivative with repect to $\hat{z}$ yields the constant of best fit, the median of the residuals.

```
class AbsoluteDeviationLossFunctionBDT
  Init(events)  # set initial predictions to the median
    median = Fit(events)
    for(event in events)
      event.predictedValue+=median

  SetTargets(events)
    for(event in events)
      event.target = Target(event)

  Target(event)  # sign of residual
    return sign(event.trueValue - event.predictedValue)

  Fit(events)    # median
    for(event in events)
      residuals.add(event.trueValue-event.predictedValue)
```

```
    return median(residuals);
```

4.1.4 HuberLossFunctionBDT

This loss function inherits LossFunctionBDT and implements the functions with the appropriate behavior for this error metric. Huber has one parameter, the quantile defining the cutoff residual, δ. The quantile parameter and the distribution of events residuals determine the cutoff residual, δ. For a quantile of 0.7, δ would be $|z-\hat{z}|$ where 70% have a $|z-\hat{z}|$ value less than or equal to δ.

$$\text{Huber} \equiv \sum_{i=1}^{N} f(z^i, \hat{z}^i; \delta) \tag{15}$$

$$f(z, \hat{z}; \delta) = \begin{cases} \frac{1}{2}(z-\hat{z})^2 & |z-\hat{z}| \leq \delta \\ \delta|z-\hat{z}| - \frac{1}{2}\delta^2 & otherwise \end{cases}$$

Taking a derivative with respect to the ith predicted value yields sign$(z-\hat{z})$min$(|z-\hat{z}|,\delta)$. Setting $\hat{z}$ to a constant for all i and taking a derivative with repect to $\hat{z}$ yields the constant of best fit, the shifted median. The shifted median is between the mean and the median. The exact form is shown in the code below. As the quantile of the cutoff goes to 100% the shifted median approaches the mean. As the quantile goes to 0% the shifted median approaches the median. So Huber can be tought of as a mixture of Least Squares and Absolute Deviation, where the quantile determines the mixing. With this new verison of Huber the user can choose the quantile value, giving them greater control over their focus. This was not possible with the hard-coded loss function in the baseline version of TMVA.

```
class HuberLossFunctionBDT

  fQuantile        # set by user
  fResidualCutoff  # delta

  Init(events)  # set initial predictions to the shifted_median
    shifted_median = Fit(events)
    for(event in events)
      event.predictedValue+=shifted_median

  SetTargets(events)
    for(event in events)
      event.target = Target(event)

  Target(event)  # sign of residual
    residual = event.trueValue - event.predictedValue
    return sign(residual)*min(fResidualCutoff, abs(residual))

  Fit(events)    # shifted_median
```

```
residualMedian = getResidualMedian(events)
shift = 0
for(event in events)
  residual = event.trueValue - event.predictedValue
  diff = residual-residualMedian
  # shift will be average of differences from median
  # except that we discount diff to the residual cutoff
  # if it is too large.
  shift+=1/events.size()*sign(diff)*min(fResidualCutoff,abs(diff))
return (residualMedian + shift)
```

4.1.5 Results

With the the loss functions now abstracted in TMVA, all three were tested on a standard regression sample to check their behavior. Absolute deviation weights the tails the least so it should perform the best in the core of the residuals and the worst in the tails. Least Squares weights the tails the most of the three loss functions so it should do the best there and the worst in the core of the residuals. Huber with the quantile set to 70% behaves like a mixture of the two.

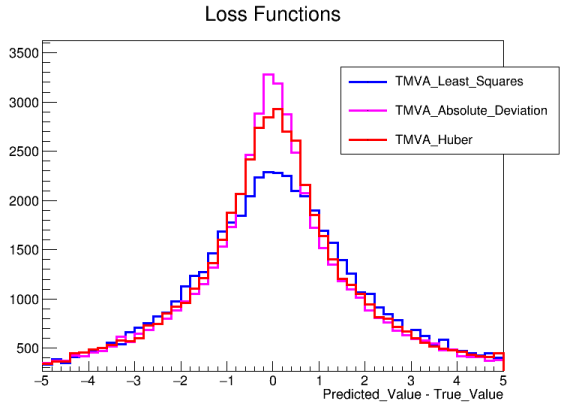

Figure 4: The loss function performance in the core.

As can be seen in Figure 4 the loss functions behave as expected in the core. In Figure 5 the loss functions behave as expected in the tails.

These loss functions are now part of TMVA as of ROOT 6. A Jupyter notebook illustrating the new loss function capabilities is available on SWAN here, http://swan.web.cern.ch/content/machine-learning. The Jupyter notebook also provides the first TMVA regression example on SWAN. In addition, the TMVA Users Guide has been updated documenting the new capabilities, see Figure 6. The updated guide will be available in the next development version of ROOT and available to download.

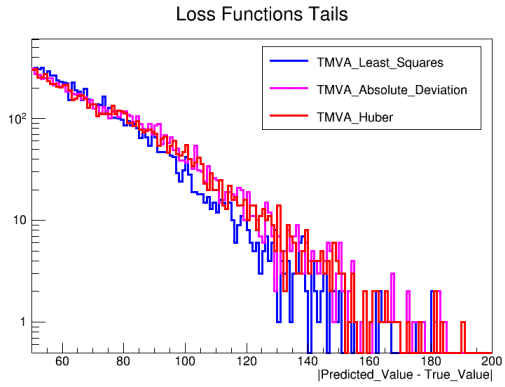

Figure 5: The loss function performance in the tails.

Option	Array	Default	Predefined Values	Description
`NTrees`	–	800	–	Number of trees in the forest
`MaxDepth`	–	3	–	Max depth of the decision tree allowed
`MinNodeSize`	–	5%	–	Minimum percentage of training events required in a leaf node (default: Classification: 5%, Regression: 0.2%)
`nCuts`	–	20	–	Number of grid points in variable range used in finding optimal cut in node splitting
`BoostType`	–	`AdaBoost`	`AdaBoost`, `RealAdaBoost`, `Bagging`, `AdaBoostR2`, `Grad`	Boosting type for the trees in the forest (note: AdaCost is still experimental)
`RegressionLossFunctionBDTG`		`Huber`	`Huber`, `LeastSquares`, `AbsoluteDeviation`	Loss function for regression when using Grad boosting.
`HuberQuantile`	–	0.7	–	The quantile that divides the core residuals from the tail residuals in the Huber loss function.

Figure 6: The relevant portion of the updated TMVA Users Guide.

4.2 Multi-threading the Training

The ROOT object TThreadExecutor handles the multi-threading in ROOT and works with Intel's open source multi-threading package Thread Building Blocks underneath. The primary targets to parallelize in the code were those processes with loops over the number of events, since these loops are generally the most computationally intensive processes in terms of time. The loop over the events was parallelized by splitting the iteration over the events into sections. The code gets the number of available processors and splits the events into that many chunks. These chunks then run in parallel.

Some portions of the code are parallelized by replacing the loop over the features with a

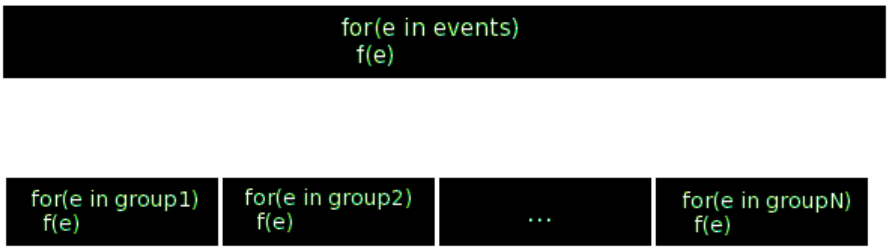

Figure 7: The loop over the data is split into chunks. The chunks run in parallel. The threads are then assigned to the pieces.

pool of tasks, one task per feature. For instance, if there are three features, three tasks will be created: one for feature 0, one for feature 1, and one for feature 2. The available threads are then assigned to the tasks, and the threads run in parallel. This method is used mainly when there is a loop over the bins within a loop over the features. Such parallelization is helpful when features*bins is on the order of the data.

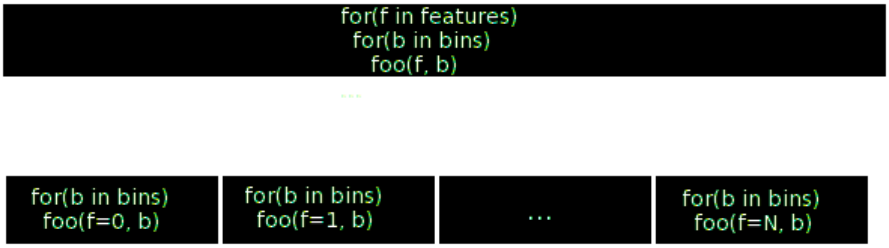

Figure 8: The loop over the features is replaced by a pool of tasks, one for each feature. The threads are then assigned to the tasks.

4.2.1 Outlining the Parallelization in the Code

The relevant portions of the code were assessed for their viability in multi-threading and prioritized based upon feasibility and computation time. The parallelization of the algorithm is outlined below. The parallelized processes are marked with a p and those that were not parallelized are marked with a !p. Those processes with parallelized subprocesses are marked with a -p.

(!p) init-events Initializes some loss function parameters and some values for the events. Happens once before building the set of trees. Since this happens once and not during every tree it is not a large contribution to the time of the algorithm. This was not targeted for parallelization.

(!p) build-forest-loop Creates the trees of the forest looping until maxtrees is reached. Each tree depends on the results of the tree before, so this loop itself can't be parallelized.

(-p) build-tree Recursively builds the nodes of the tree. Building the next node depends on the node before. This function itself can't be parallelized, but some subprocesses are.

- **(p) build-sum** Loops over the events to calculate xmin, xmax for the features and some needed sums. Parallelized by chunking the loop over the training events.
- **(-p) build-train** The subprocesses are parallelized.
  - **(p) build-train-init-hists** Loops over bins within a feature loop to initialize the histograms. No loop over events, but parallelized by replacing the feature loop with one task per feature and assigning the pool of threads to the tasks.
  - **(p) build-train-fill-hists** Loops over features within an event loop to fill the histograms for each feature. Parallelized by chunking the loop over the events.
  - **(p) build-train-accumulate-hists** Loops over bins within a feature loop to turn the histograms into cumulative histograms. No loop over events, but parallelized by replacing the feature loop with one task per feature and assigning the pool of threads to the tasks.
  - **(p) build-train-calc-best-error** Loops over the bins within a feature loop to calculate the best error reduction for each split point saving the max info. No loop over events, but parallelized by replacing the feature loop with one task per feature and assigning the pool of threads to the tasks.
- **(!p) build-filter** Loops over the events. Events from the mother node are appended to the leftSample and rightSample vectors depending on the feature value and the cut. Two processes cannot append to the same vector at the same time, so this was not parallelized.

(!p) boost-leaf Loops over the events to add them to the correct terminal node collection. Again, events are filtered from the main vector into multiple vectors, this time the leaf vectors. Multiple processes can't append to the same vectors at the same time.

(!p) boost-fit Sets the fit constant in each terminal node. This is a very small portion of the total time and was not targeted for parallelization.

(p) boost-update-predictions Loops over the events updating the predictions. Parallelized by chunking the loop over the training events.

(p) boost-update-targets Loops over the events setting the new targets for the next tree. Parallelized by chunking the loop over the training events.

4.2.2 Timing the Parallelized Algorithm

The multithreaded algorithm and the baseline algorithm processes were timed in order to compare the improvements and assess the behavior of the parallelized processes. The timing for 10 trees, depth 4, nbins 100, 16 features and 1 million events are presented here. These are

very typical settings for a regression problem. The times represent the net time taken by the process over the 10 trees, and these times are presented as a function of the number of CPUs = (1,2,4,8,16,32). The timing study was performed on a private server, olhswep13.cern.ch, with 52 available cores.

The baseline is in blue and the mean time over the 6 nCPU runs is used as its plot. First the decision tree algorithm processes are presented, then the boosting processes, and finally the net timing over the 10 trees.

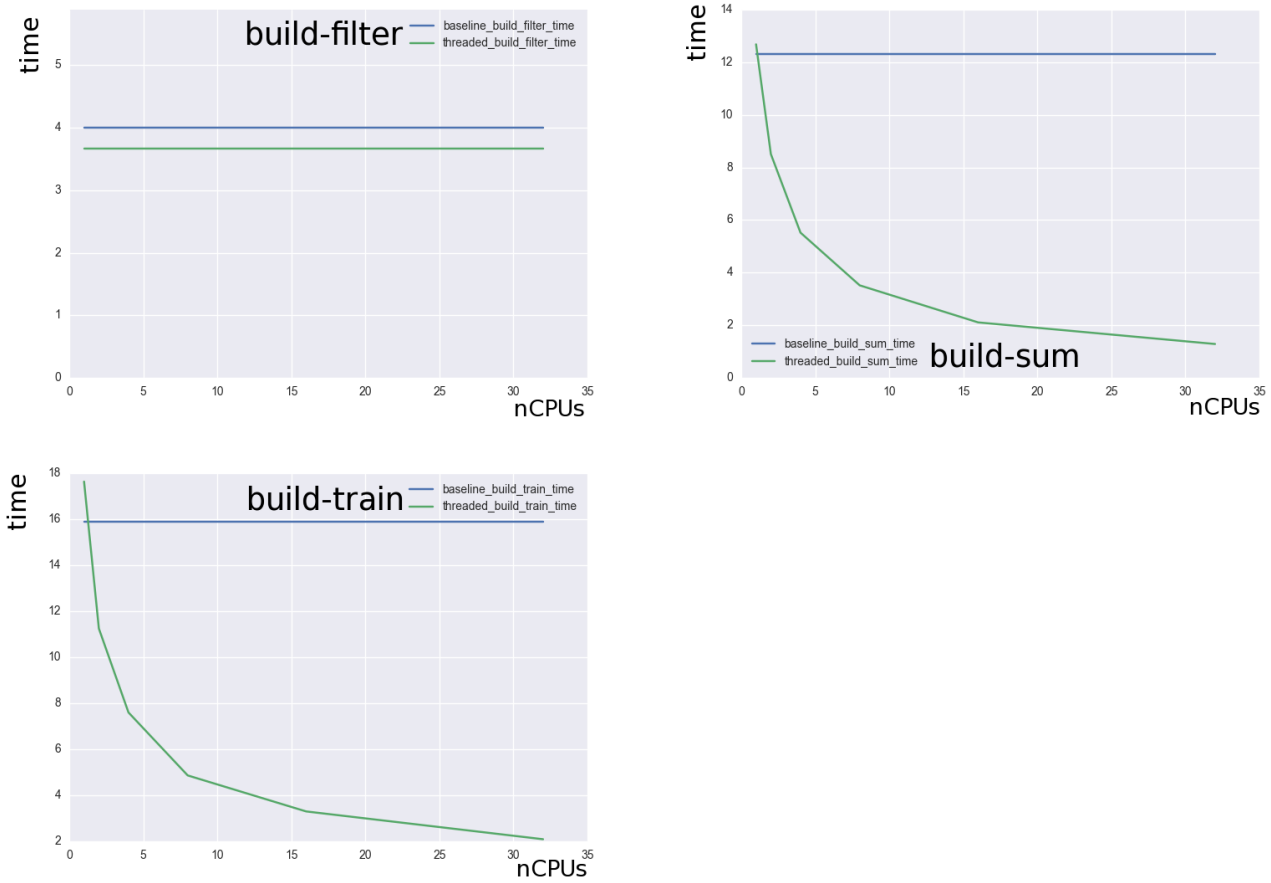

Figure 9: Timing improvements for the decision tree (build-tree) processes. The baseline is in blue and the parallelized versions are in green.

The parallelization in both the boosting and the tree building behaves as expected with the time almost halving every time the processors are doubled. More accurately, time reduces by a factor of 1.6 every doubling.

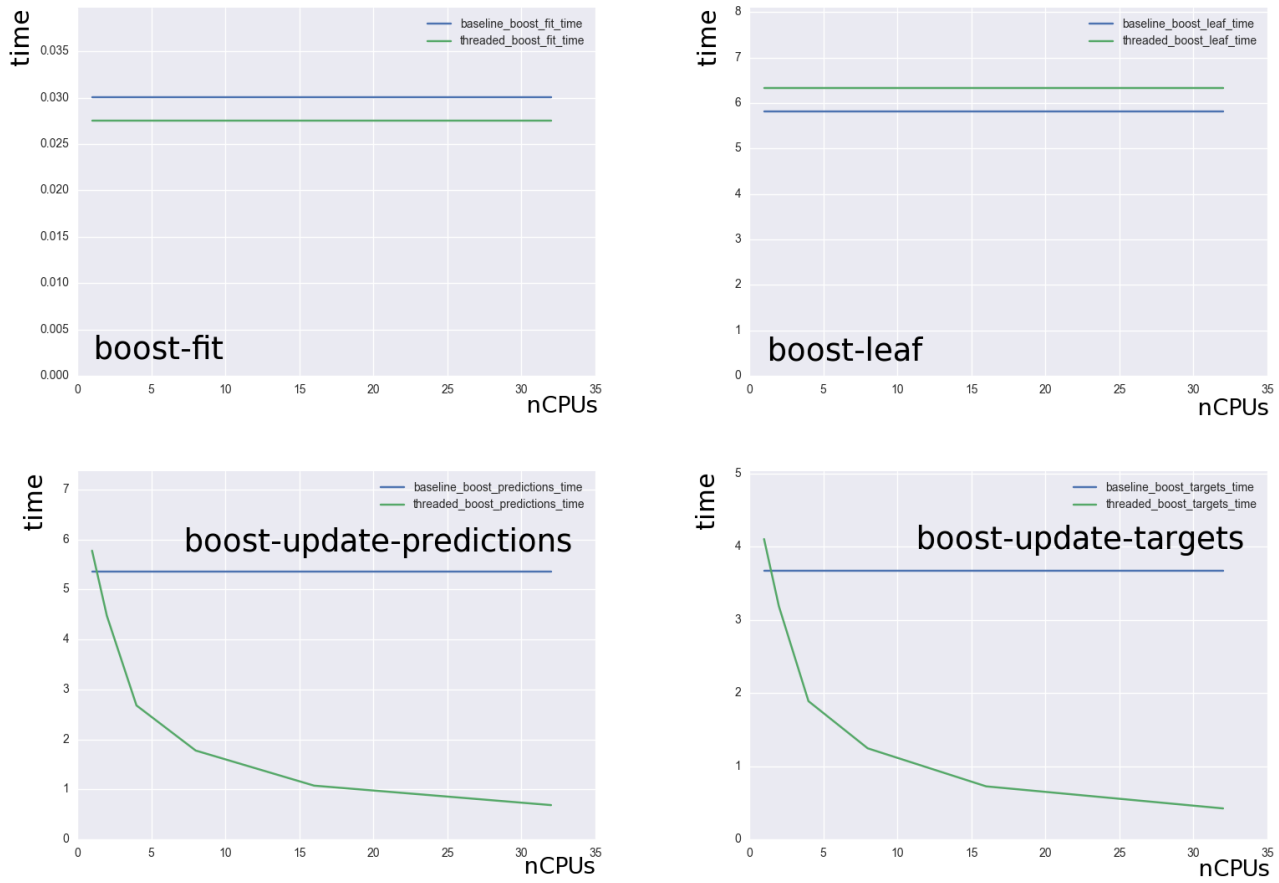

Figure 10: Timing improvements for the boosting processes. The baseline is in blue and the parallelized versions are in green.

The net timing for the multithreaded verison of the BDT algorithm asymptotes at around 15 seconds, Figure 11, due mainly to the processes that were not parallelized. If clever methods of parallelization exist for those processes, further gains are possible. While BDTs do not naturally lend themselves to parallelization like say a random forest, targeting the individual processes has provided important gains. Looking at Figure 12 the multithreaded algorithm is roughly 1.7x faster with four cores and a maximum of about 3x faster.

4.2.3 Multi-threading Conclusions

With 4 cores the parallelized version is about 1.7x faster, and with 16 cores it's about 2.6x faster. With the same settings and 100,000 events the performance gains are very similar at 1.5x with 4 cores and 2.4x with 16 cores. The gains top off around 3x. While the loss functions have already been incorporated, the parallelized version will be available in an upcoming ROOT release.

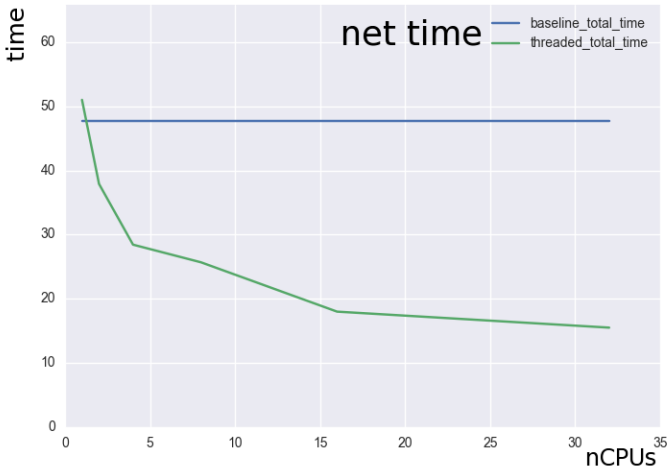

Figure 11: Net timing for ten trees. The baseline is in blue and the multithreaded version is in green. The multithreaded time asymptotes at a constant value due to the nonparallelized processes.

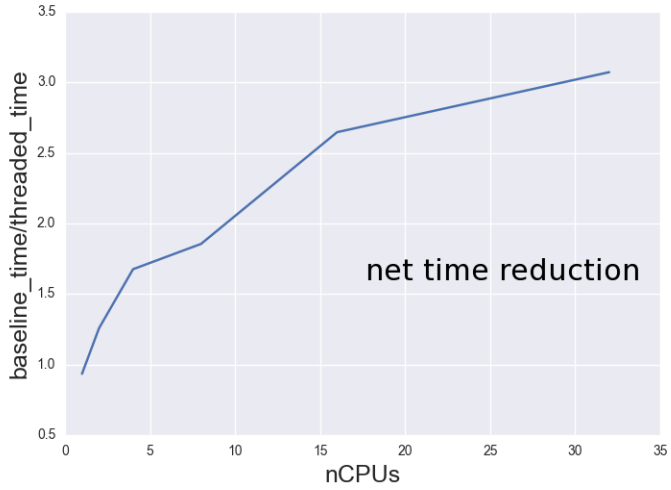

Figure 12: The reduction factor in time as a function of nCPUs. With 16 CPUs the multithreaded algorithm is roughly 2.6 times faster.

4.3 Debugging the Evaluation

In many cases the evaluation of the events in the BDTs was taking longer than the training, however in terms of complexity the operations are far fewer. To evaluate one event on a single tree of depth 4, the event will have to go through 3 comparisons. This yields 12 comparisons per tree. For ten trees this is 120 comparisons. Multiplying by 10 to account for overhead gives 1,200 comparisons total multiplying by 1 million events that's 1.2 billion operations. A 1.2 GHz processor can perform 1.2 billion operations per second, so the evaluation should take roughly 1 second. Given this rough estimate it's safe to assume the evaluation should not take 15 minutes, which was the case for 1 million events, 10 trees, and a depth of 4.

Given this back of the envelope estimate, TMVA's evaluation appeared to be about two orders of magnitude off.

Comparing the BDT evaluation in TMVA to another C++ BDT package revealed that TMVA was indeed about 500x slower. As it turns out the evaluation was drawing the progress bar for every event evaluation, which means that for 1 million events it was drawing the progress bar 1 million times. With this bug fixed the evaluation time for 1 million events was reduced from 922 seconds to 2 seconds. the order of magnitude expected.

While the multi-threading provided a speedup in the training somewhere between 1.5 and 3x, fixing this simple bug provided a speed up of 460x for the evaluation. This fix will be available along with the parallelization.

5 Conclusion

After all of the improvements, the BDT evaluation in ROOT's TMVA (for 1 million events) is 460x faster, the training is 1.5x to 3x faster, and there are now three standard loss functions to choose from instead of one. To make sure the changes didn't break anything, the predictions of the BDTs with the new loss functions and parallelization were compared to the baseline – TMVA's BDTs before any modifications. The trees with the modifications modeled the training set the same way as the baseline for various values of ntrees, depth, nbins, features, ncpus, and ntrainingdata.

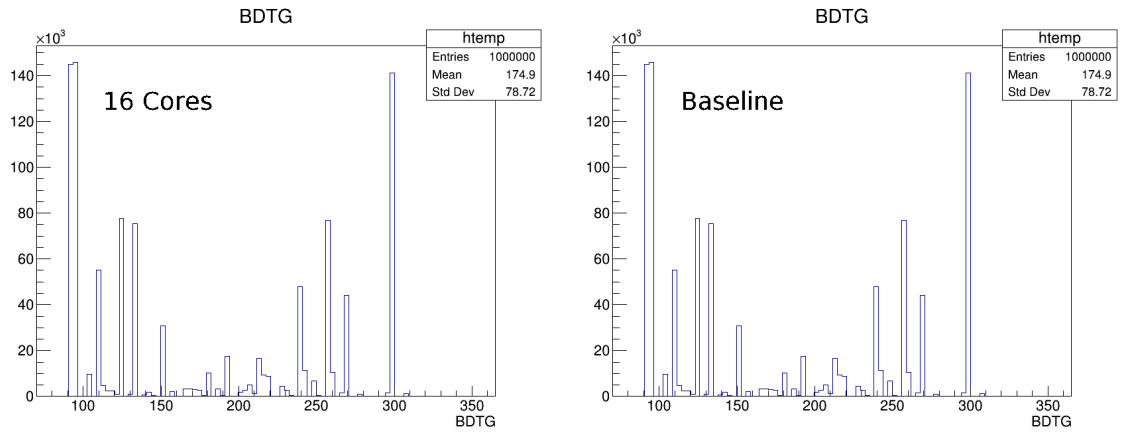

Figure 13: The evaluation of the training on the training set for TMVA with all of the changes compared to the baseline (before any changes). The results for 2 trees, depth 4, nbins 20, 16 vars, and 1,000,000 training events the results are identical.

Moreover, the Users Guide has been updated with the new features and there is a Jupyter notebook illustrating these capabilities on SWAN. The notebook is available at http://swan.web.cern.ch/content/machine-learning. The loss functions are now available in ROOT, but the parallelization, Users Guide, and evaluation bug fix, while completed, will be available in the next development release.

The loss function improvements may be found here.

`https://root.cern.ch/doc/master/LossFunction_8h_source.html`

https://root.cern.ch/doc/master/LossFunction_8cxx_source.html
https://root.cern.ch/doc/master/MethodBDT_8h_source.html
https://root.cern.ch/doc/master/MethodBDT_8cxx_source.html

ROOT and TMVA are found here.

https://root.cern.ch/
https://root.cern.ch/tmva

The new loss functions, parallelization, and quick evaluation are already being used to develop the next round of momentum assignment in the Endcap Muon Track Finder (EMTF), part of the Level 1 Trigger at the Compact Muon Solenoid (CMS) experiment at CERN. The improved focus on low momentum events along with the quick turn around from the parallelization and evaluation have allowed development to proceed quickly and with great efficacy, already showing tentative improvements in the momentum assignment over the baseline algorithm. With continued development, the updated EMTF momentum assignment will enable the experiment to keep more of the interesting data at higher luminosities. In addition, the updated BDTs will enable rapid development and improved results for other regression tasks in many physics analyses and other detector implementations.

References

[1] Friedman, Jerome H. Greedy Function Approximation: A Gradient Boosting Machine. The Annals of Statistics, vol. 29, no. 5, 2001, pp. 11891232. www.jstor.org/stable/2699986.

[2] TMVA Users Guide. http://tmva.sourceforge.net/docu/TMVAUsersGuide.pdf

[3] Trevor Hastie, Robert Tibshirani, Jerome Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. 2009. https://statweb.stanford.edu/ tibs/ElemStatLearn/

[4] ROOT. https://root.cern.ch/

[5] TMVA. https://root.cern.ch/tmva