BDTLib
A Gradient Boosted Decision Tree Package

By Andrew Carnes

UF
UNIVERSITY of FLORIDA

Intro

- BDTLib is a library implementing the Gradient Boosted Decision Tree machine learning algorithm
- Written in C++
    - Uses some basic ROOT objects like TString, TMath
- Originally developed to improve the momentum assignment in the Muon Trigger for CMS
- Code available at https://github.com/acarnes/bdt
- Detailed example with comments available at link below

https://github.com/acarnes/bdt/blob/master/examples/BasicTrainAndTest.cxx

Outline

- Brief BDT Algorithm Overview
- Algorithm Parameters
- Package Features
- Results in Real Applications
- Simple example showing how to run the code
- Backup Slides
    - Rough diagram of the code
    - References

Brief BDT Algorithm Overview

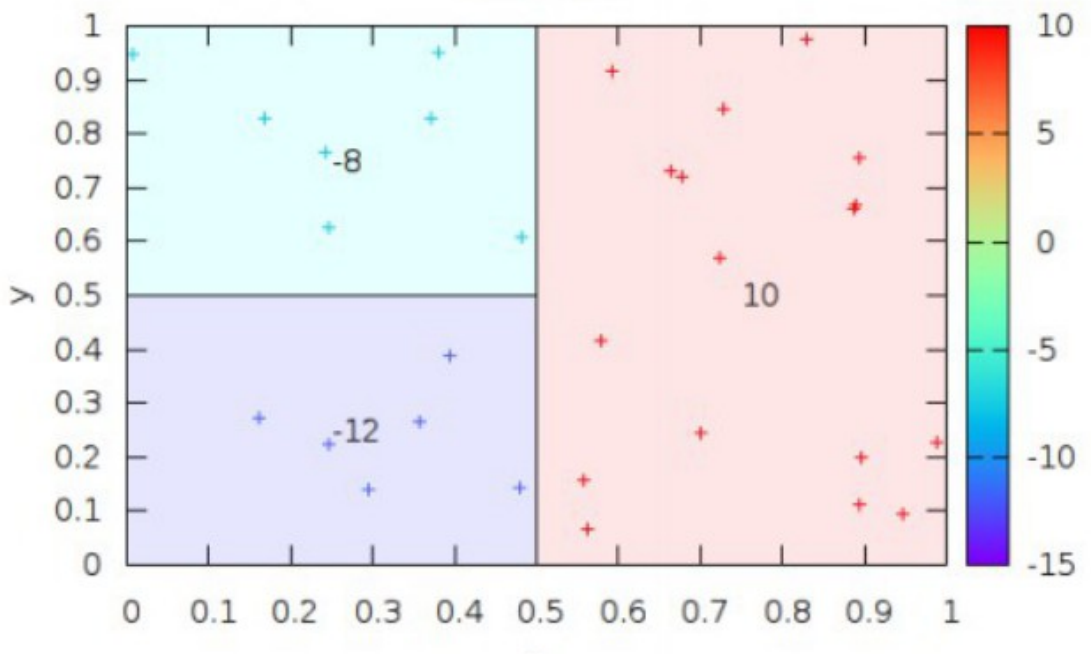


Fig 1. A decision tree with 3 terminal nodes

A Single Decision Tree

- Breaks up feature space into discrete regions using hyperplanes
- Fits a constant to each region
- The regions are greedily chosen to minimize a given Loss Function (a differentiable measure of the error)
- May be viewed as a series of decisions (shown below)

Boosting

- Make one tree, add another tree that corrects the predictions of the first
- Add another tree that corrects the net prediction of the first and second
- Continue the process
- End up with a collection of trees (Forest) and a net prediction
- $F(x) = T_0(x) + T_1(x) + T_2(x) + \ldots + T_N(x)$

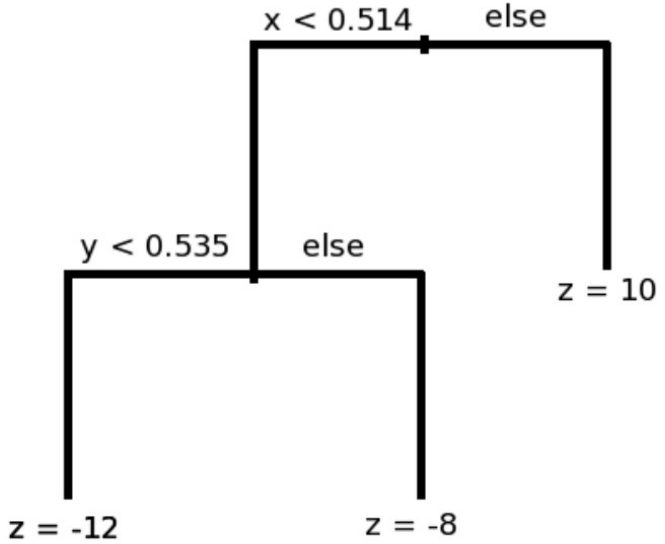


Fig 2. The same tree represented as a series of decisions

Algorithm Parameters

- **Learning Rate (Shrinkage)**
  - How quickly the algorithm heads towards the minimum
- **Number of Terminal Nodes**
  - The number of regions to break the feature space into
  - More nodes increases the complexity of the fit by increasing the complexity of each tree
- **Number of Trees**
  - The total number of trees in the forest
  - More trees increases the complexity of the fit by adding more trees
- **Loss Function**
  - A function that measures the error of the fit
  - Must be differentiable with respect to the current prediction
  - Determines shape of the regions in feature space and the constant value fit in each region for each tree in the forest

Package Features

- **Feature variable ranking**
- **A variety of loss functions** [**Only Huber in TMVA**]
  - Least Squares (R)
  - Huber (R)
  - Least Absolute Deviation (R)
  - Percent Error (R)
  - Adaboost (C)
    - untested, in development
- **Add your own loss function easily**
  - Focus on the types of events you consider important
- **Regression (R)**
- **Classification (C) (Recently devloped, untested, may crash...)**
- **Easy to use, well commented code**
- **Store the trees into xml and read the stored trees from xml**

Results in Real Applications

- BDTLib outperformed TMVA's GBDT algorithm when assigning momentum in the Level 1 Muon Triggers
    - In this study ...
    - TMVA incorrectly assigned a high momentum to low momentum muons at ~2x the rate of BDTLib (See rate reduction in backup slides)
    - Improvement over TMVA's GBDT came from BDTLib's versatility in the Loss Function (Huber vs Least Squares respectively)
- Performs regression comparably to TMVAs GBDT algo with the same parameter settings, same Loss Function
    - Gaussian resolution of the errors for the algorithms were within 1% of each other in the momentum assignment exercise

Simple Example

Detailed example with comments available at link. Excerpt detailed here.
https://github.com/acarnes/bdt/blob/master/examples/BasicTrainAndTest.cxx

```
void buildAndEvaluateForest()
{
  // Fundamental settings for the regression.
  Int_t nodes = 20;
  Int_t trees = 64;
  Double_t lr = 0.3;
  LossFunction* lf = new LeastSquares();

  bool saveTrees = false;  // Whether to save the trees from the regression into a directory specified later.
  TString treesDirectory("../trees/"); // Where to save the trees.

  std::vector<Event*> trainingEvents = std::vector<Event*>(); // The training and testing events.
  std::vector<Event*> testingEvents = std::vector<Event*>();

  loadTrainingEvents(trainingEvents, 10000);
  loadTestingEvents(testingEvents, 10);

  Forest* forest = new Forest(trainingEvents);
  forest->doRegression(nodes, trees, lr, lf, treesDirectory, saveTrees);  // Do the regression and save the trees.

  std::vector<Int_t> rank; // Rank the variable importance and output it to the screen.
  forest->rankVariables(rank);

  forest->predictEvents(testingEvents, trees);  // Make predictions on the test set using the forest that was just created

  for(unsigned int i=0; i<testingEvents.size(); i++) // Output information about the events
  {                                                  // Look at the predicted values
     Event* e = testingEvents[i];
     std::cout << "===== EVENT: " << e->id << " =======" << std::endl;
     e->outputEvent();
     std::cout << std::endl;
  }
  delete forest;
}
```

Backup Slides

- Basic outline of the code structure for the library
- References
- Rate Reduction

Code Structure

Forest.h

```
std::vector<Event*> events;
std::vector<Tree*> trees;

void rankVariables(std::vector<int>& rank);
void doRegression(Int_t nodeLimit, Int_t treeLimit,
  Double_t learningRate, LossFunction* l,
  const char* savetreesdirectory, bool saveTrees);

void predictEvents(std::vector<Event*>& eventsp, Int_t trees);
Various other functions...
```

Tree.h

```
Node *rootNode;
std::list<Node*> terminalNodes;
Int_t numTerminalNodes;
Double_t rmsError;

Various functions ...
```

Event.h

```
Double_t trueValue;
Double_t predictedValue;

// Sort the events by data[sortingIndex]
// just set the sorting index
static Int_t sortingIndex;

// Uniquely identify each event
Int_t id;

// data[0] is a special value, the target.
// Load this with the true value for most purposes.
// data[1] -> data[N] are the feature variables,
// load the feature vars into these slots
std::vector<Double_t> data;

Various functions...
```

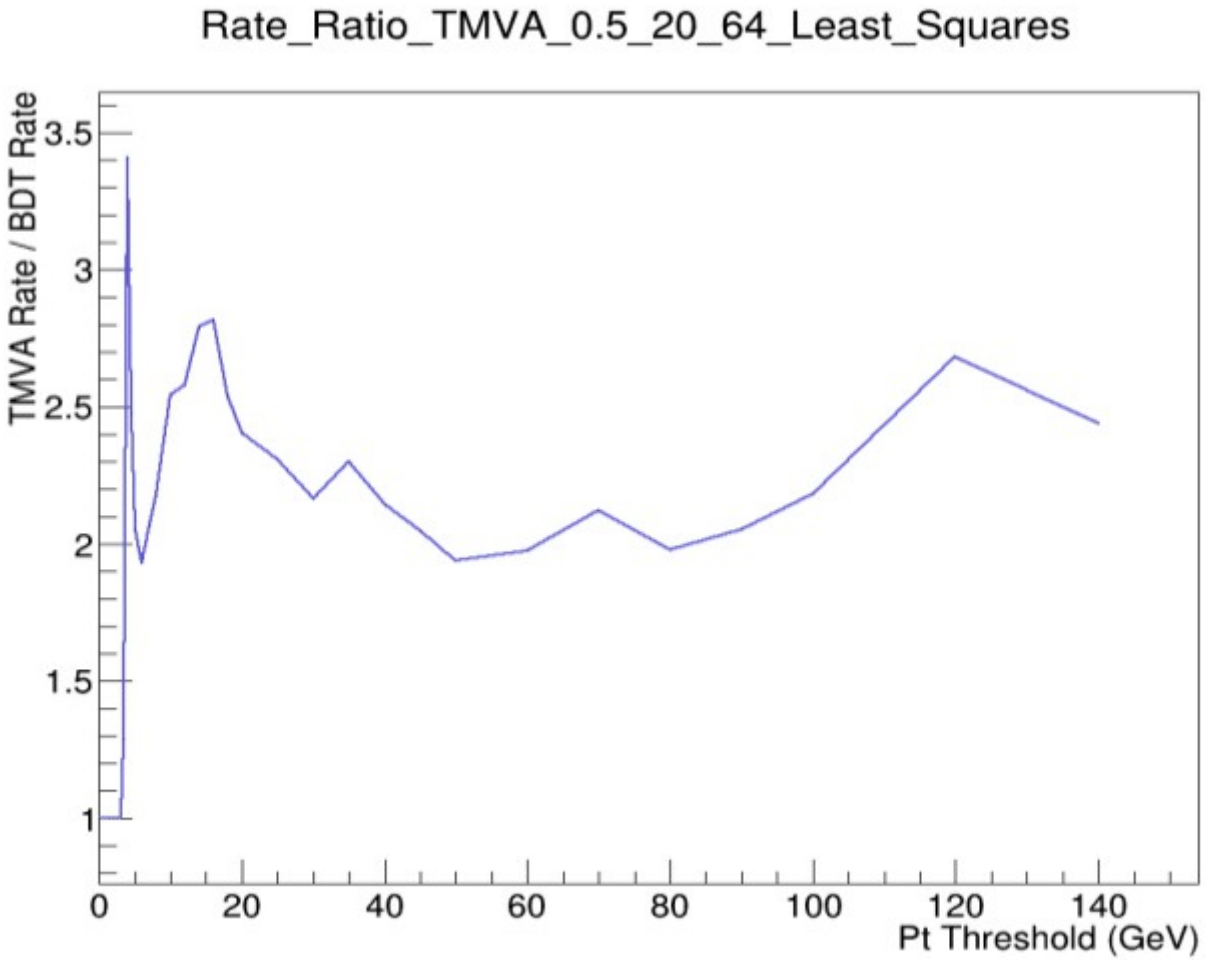
Node.h

```
std::string name;

Node *leftDaughter;
Node *rightDaughter;
Node *parent;

Double_t splitValue;
Int_t splitVariable;

Double_t errorReduction;
Double_t totalError;
Double_t avgError;

Double_t fitValue;
Int_t numEvents;

std::vector< std::vector<Event*> > events;

Various functions...
```

References

- Friedman, Jerome H. "Greedy function approximation: a gradient boosting machine." Annals of statistics (2001): 1189-1232.

Rate Reduction



- Rate shows how many muons were predicted over some Pt threshold
- Both TMVA and BDTLib correctly predicted the momentum of muons with true Pt >= the Pt thresholds with 98% efficiency
- However TMVA predicted 2.8x more muons with Pt >= 20 GeV
- All of this means it incorrectly assigned a momentum a high momentum (>= 20 GeV) at ~2.8x the rate of BDTLib
- On average TMVA incorrectly assigned high Pt at 2.2x the rate of BDTLib
- Convoluted way of measuring things but standard for the Trigger