Boosted Decision Tree

Ryan De Los Santos

3-18-24

Contents

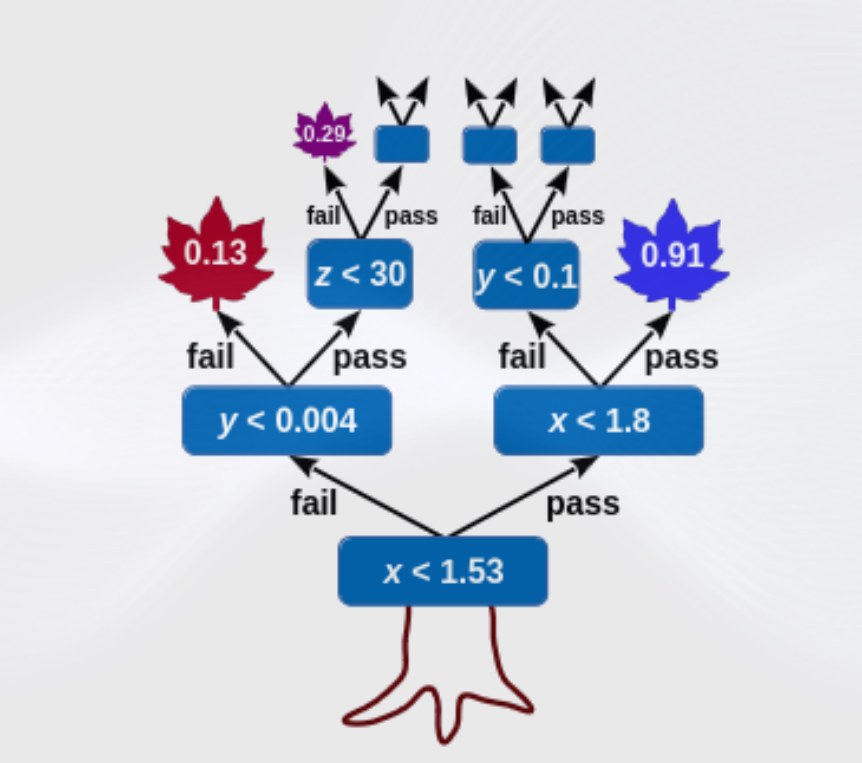
1 Boosted Decision Tree Background **1**

2 Setting Up Boosted Decision Tree **2**
2.1 Data Preprocessing 2
2.2 BDT Creation 4

1 Boosted Decision Tree Background

As can be seen from the previous assignment, the usual way to select a cut is to make signal to background ratio plots. While this process may seem easy for only one of two variables, analysis almost never deal with optimizing only one or two variables. This can cause issues since making cuts on one variable can affect the distributions of other variables, making the cut optimization much more complicated.

This is where we bring in neural networks and allow them to perform their own magic to optimize the cuts for us. A decision tree creates a series of conditions that attempts to optimize the seperation between two classes. In our case, these two classes are the signal and background. For a visual representation on what a boost decision tree does see 1



2 Setting Up Boosted Decision Tree

2.1 Data Preprocessing

We now want to train a decision tree to implement the cuts that we decided on manually in the last section.

We start off first by formatting our data

```
data_for_BDT = {} # define empty dictionary to hold
↪  dataframes that will be used to train the BDT
BDT_inputs = ['lep_pt_1','lep_pt_2'] # list of features
↪  for BDT
for key in data: # loop over the different keys in the
↪  dictionary of dataframes
    data_for_BDT[key] = data[key][BDT_inputs].copy()
```

Neural networks are usually organized into 2D arrays where the rows represent the number of samples you want to pass to the model and the columns

represent the number of features that your data has. For example if we have 500 events and each event contains information about the momentum, mass, and charge of a particle. Then our data should be a 500x3 array.

```
all_MC = [] # define empty list that will contain all
↪  features for the MC
for key in data: # loop over the different keys in the
↪  dictionary of dataframes
    if key!='data': # only MC should pass this
        all_MC.append(data_for_BDT[key]) # append the MC
        ↪  dataframe to the list containing all MC
        ↪  features
X = np.concatenate(all_MC) # concatenate the list of MC
↪  dataframes into a single 2D array of features, called X

all_y = [] # define empty list that will contain labels
↪  whether an event in signal or background
for key in data: # loop over the different keys in the
↪  dictionary of dataframes
    if key!=r'$H \rightarrow ZZ \rightarrow
    ↪  \ell\ell\ell\ell$' and key!='data': # only
    ↪  background MC should pass this
        all_y.append(np.zeros(data_for_BDT[key].shape[0]))
        ↪  # background events are labelled with 0
all_y.append(np.ones(data_for_BDT[r'$H \rightarrow ZZ
↪  \rightarrow \ell\ell\ell\ell$'].shape[0])) # signal
↪  events are labelled with 1
y = np.concatenate(all_y) # concatenate the list of lables
↪  into a single 1D array of labels, called y
```

When training a nueral network, one of the major things that we want to be aware of is overtraining. This is the phenomenon where a model learns some specific data very well, but the model fails to generalize to different data. This make the model useless and as such we need to take extra precaution.

One of the ways, we can check that we are not overtraining our model is by seperating our datasets into training and testing datasets. This way, the model can train on a subset of the data, then we will evaluate it on data that it has never seen before. If the model is overtrained then we would expect

it to perform well on the training dataset but then perform poorly on the testing dataset. Luckily, a python module allows us to correctly split our dataset while ensuring that everything is still random.

```
from sklearn.model_selection import train_test_split

# make train and test sets
X_train,X_test, y_train,y_test = train_test_split(X, y,
                                                  test_size=0.33,
                                                  random_state=4
                                                  ↪  92
                                                  ↪  )
```

2.2 BDT Creation

We are now at a position where we can create our boosted decision tree. We will make use of the functions already provided by scikitlearn (DecisionTreeClassifier and AdaBoostClassifier).

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

dt = DecisionTreeClassifier(max_depth=2) # maximum depth of
↪  the tree
bdt = AdaBoostClassifier(dt,
                         algorithm='SAMME', # SAMME discrete
                         ↪  boosting algorithm
                         n_estimators=12, # max number of
                         ↪  estimators at which boosting is
                         ↪  terminated
                         learning_rate=0.5) # shrinks the
                         ↪  contribution of each classifier by
                         ↪  learning_rate

start = time.time() # time at start of BDT fit
bdt.fit(X_train, y_train) # fit BDT to training set
elapsed = time.time() - start # time after fitting BDT
print("Time taken to fit BDT: "+str(round(elapsed,1))+"s") #
↪  print total time taken to fit BDT
print(bdt)
```

Now that the model is trained, we can assess its performance by running the model over the test dataset.

```
from sklearn.metrics import classification_report,
↪  roc_auc_score
y_predicted = bdt.predict(X_test) # get predicted y for test
↪  set
print (classification_report(y_test, y_predicted,
                             target_names=["background",
                             ↪  "signal"]))
print ("Area under ROC curve for test data:
↪  %.4f"%(roc_auc_score(y_test,
↪  bdt.decision_function(X_test))) )
```

What would you expect to see if we ran the model over the training dataset?
Modify the above code to prove that this is in fact the case