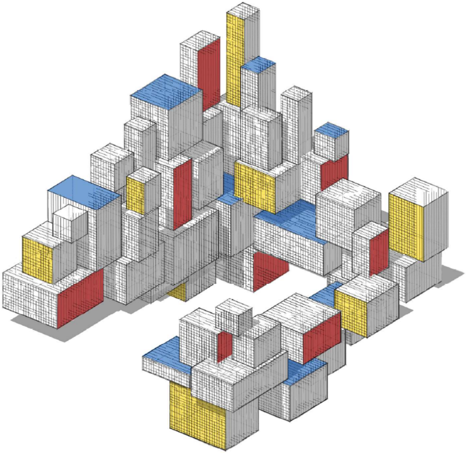
Decision Tree Construction



Purpose

In this lecture we discuss a basic, step by step, implementation of a decision tree.

Data Generation

First, we import various packages and define a function to generate the training and test data.

BasicTree.py

```
import numpy as np
from sklearn.datasets import make_friedman1
from sklearn.model_selection import train_test_split

def makedata():
   n_points = 500 # number of samples

   X, y =  make_friedman1(n_samples=n_points, n_features=5,
                          noise=1.0, random_state=100)
   return train_test_split(X, y, test_size=0.5, random_state=3)
```

The “main” method calls the **makedata** method, uses the training data to build a regression tree, and then predicts the responses of the test set and reports the mean squared-error loss.

```
def main():
  X_train, X_test, y_train, y_test = makedata()
  maxdepth = 10 # maximum tree depth
  # Create tree root at depth 0
  treeRoot = TNode(0, X_train,y_train)

  # Build the regression tree with maximal depth equal to max_depth
  Construct_Subtree(treeRoot, maxdepth)

  # Predict
  y_hat = np.zeros(len(X_test))
  for i in range(len(X_test)):
     y_hat[i] = Predict(X_test[i],treeRoot)

  MSE = np.mean(np.power(y_hat - y_test,2))
  print("Basic tree: tree loss = ",  MSE)
```

Tree Node Class

The next step is to specify a tree node as a Python class. Each node has a number of attributes, including the features and the response data (`X` and `y`) and the depth at which the node is placed in the tree. The root node has depth 0. Each node w can calculate its contribution to the squared-error training loss $\sum_{i=1}^{n} \mathbb{I}\{\boldsymbol{x}_i \in \mathcal{R}^w\}(y_i - g^w(\boldsymbol{x}_i))^2$. Note that we have omitted the constant $1/n$ term when training the tree.

```
class TNode:
   def __init__(self, depth, X, y):
      self.depth = depth
      self.X = X   # matrix of features
      self.y = y   # vector of response variables
      # initialize optimal split parameters
      self.j = None
      self.xi = None
```

```python
    # initialize children to be None
     self.left = None
     self.right = None
     # initialize the regional predictor
     self.g = None

  def CalculateLoss(self):
     if(len(self.y)==0):
        return 0

     return np.sum(np.power(self.y - self.y.mean(),2))
```

Tree Building Method

```
def Construct_Subtree(node, max_depth):
    if(node.depth == max_depth or len(node.y) == 1):
        node.g  = node.y.mean()
    else:
        j, xi = CalculateOptimalSplit(node)
        node.j = j
        node.xi = xi
        Xt, yt, Xf, yf = DataSplit(node.X, node.y, j, xi)

        if(len(yt)>0):
            node.left = TNode(node.depth+1,Xt,yt)
            Construct_Subtree(node.left, max_depth)

        if(len(yf)>0):
            node.right = TNode(node.depth+1, Xf,yf)
            Construct_Subtree(node.right, max_depth)

    return node
```

Splitting the Data

The tree building method requires an implementation of the `CalculateOptimalSplit` function.

To start, we implement a function `DataSplit` that splits the data according to $s(\boldsymbol{x}) = \mathbb{I}\{x_j \leqslant \xi\}$.

```
def DataSplit(X,y,j,xi):
    ids = X[:,j]<=xi
    Xt  = X[ids == True,:]
    Xf  = X[ids == False,:]
    yt  = y[ids == True]
    yf  = y[ids == False]
    return Xt, yt, Xf, yf
```

The `CalculateOptimalSplit` method runs through the possible splitting thresholds ξ from the set $\{x_{j,k}\}$ and finds the optimal split.

```
def CalculateOptimalSplit(node):
    X = node.X
    y = node.y
    best_var = 0
    best_xi = X[0,best_var]
    best_split_val = node.CalculateLoss()
    m, n  = X.shape

    for j in range(0,n):
        for i in range(0,m):
            xi = X[i,j]
            Xt, yt, Xf, yf = DataSplit(X,y,j,xi)
            tmpt = TNode(0, Xt, yt)
            tmpf = TNode(0, Xf, yf)
            loss_t = tmpt.CalculateLoss()
            loss_f = tmpf.CalculateLoss()
            curr_val =  loss_t + loss_f
            if (curr_val < best_split_val):
                best_split_val = curr_val
                best_var = j
                best_xi = xi
    return best_var,  best_xi
```

Prediction

Finally, we implement the recursive method for prediction.

```
def Predict(X,node):
    if(node.right == None and node.left != None):
        return Predict(X,node.left)

    if(node.right != None and node.left == None):
        return Predict(X,node.right)

    if(node.right == None and node.left == None):
        return node.g
    else:
        if(X[node.j] <= node.xi):
            return Predict(X,node.left)
        else:
            return Predict(X,node.right)
```

Putting it All Together

Running the `main` function defined above gives a similar result to what one would achieve with the `sklearn` package, using the `DecisionTreeRegressor` method.

```
main()  # run the main program

# compare with sklearn
from sklearn.tree import DecisionTreeRegressor

X_train, X_test, y_train, y_test = makedata() # use the same data
regTree = DecisionTreeRegressor(max_depth = 10, random_state=0)
regTree.fit(X_train,y_train)
y_hat = regTree.predict(X_test)
MSE2 = np.mean(np.power(y_hat - y_test,2))
print("DecisionTreeRegressor: tree loss = ",  MSE2)
```

```
Basic tree: tree loss =  9.067077996170276
DecisionTreeRegressor: tree loss =  10.197991295531748
```