# CART in Haskell

Genji Ohara

June 28, 2023

## 1 Preamble

```haskell
import DataProcessing
import Numeric.LinearAlgebra
import Prelude hiding ((<>))

type Vec = Vector R
-- type Mat = Matrix R
```

## 2 Data Type Definition

### 2.1 Data Space

| | |
|---|---|
| Feature Space | $\mathcal{F} = \mathbb{R}^D$ |
| Label Space | $\mathcal{L} = \{0, 1, \ldots, L-1\}$ |
| Data Space | $\mathcal{D} = \mathcal{F} \times \mathcal{L}$ |

## 2.2 Constants

```
featureNum :: Int
featureNum = 4

labelNum :: Int
labelNum = 3
```

## 2.3 Tree Structure

### 2.3.1 Literal

```
data Literal = Literal Int Double
-- data Literal = Literal {
--     lFeatureIdx :: Int,
--     lValue :: Double
-- }

instance Show Literal where
    show (Literal i v) = "Feature[" ++ (show i) ++ "] < " ++ (show v)
```

### 2.3.2 Split

```
data Split = Split {
    sLiteral :: Literal,
    sScore :: Double
} deriving Show

instance Eq Split where
    (Split _ s) == (Split _ s’) = s == s’

instance Ord Split where
    compare (Split _ s) (Split _ s’) = compare s s’
```

### 2.3.3 Tree

```
data Tree = Leaf Int String | Node Literal Tree Tree String
-- data Tree = Leaf {label :: Int, id :: String} |
--             Node {literal :: Literal, left :: Tree, right :: Tree, id :: String}
```

# 3 Output Tree

```
instance Show Tree where
    show tree = treeToString tree 0

treeToString :: Tree -> Int -> String
treeToString (Leaf l _) depth =
    branchToString depth ++ "class: " ++ (show l) ++ "\n"
treeToString (Node literal leftTree rightTree _) depth =
    let str1 = branchToString depth ++ show literal ++ "\n"
        str2 = treeToString leftTree (depth + 1)
        str3 = branchToString depth ++ "!" ++ show literal ++ "\n"
        str4 = treeToString rightTree $ depth + 1
    in str1 ++ str2 ++ str3 ++ str4

branchToString :: Int -> String
branchToString depth = "|" ++ (concat $ replicate depth "   |") ++ "--- "
```

Listing 1: Example of CLI output

```
|--- Feature[2] < 1.9
|   |--- class: 0
|--- !Feature[2] < 1.9
|   |--- Feature[3] < 1.7
|   |   |--- Feature[2] < 4.9
|   |   |   |--- Feature[3] < 1.6
|   |   |   |   |--- class: 1
|   |   |   |--- !Feature[3] < 1.6
|   |   |   |   |--- class: 2
|   |   |--- !Feature[2] < 4.9
|   |   |   |--- Feature[3] < 1.5
|   |   |   |   |--- class: 2
|   |   |   |--- !Feature[3] < 1.5
|   |   |   |   |--- Feature[0] < 6.7
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- !Feature[0] < 6.7
|   |   |   |   |   |--- class: 2
|   |--- !Feature[3] < 1.7
|   |   |--- Feature[2] < 4.8
|   |   |   |--- Feature[0] < 5.9
|   |   |   |   |--- class: 1
|   |   |   |--- !Feature[0] < 5.9
|   |   |   |   |--- class: 2
|   |   |--- !Feature[2] < 4.8
|   |   |   |--- class: 2
```

# 4 Gini Impurity

## 4.1 Class Ratio

$$\begin{array}{lll}
\text{Label Set} & L = \{y \mid (\boldsymbol{x}, y) \in D\} & \\
\text{Label Count} & c_l(L) = \displaystyle\sum_{i \in L} \mathbb{I}[i = l], & \boldsymbol{c}(L) = \displaystyle\sum_{i \in L} \mathrm{onehot}(i) \\
\text{Class Ratio} & p_l(L) = \dfrac{c_l(L)}{|L|}, & \boldsymbol{p}(L) = \dfrac{\boldsymbol{c}(L)}{\|\boldsymbol{c}(L)\|_1}
\end{array}$$

```
labelCount :: [Label] -> Vec
labelCount = sum . (map $ oneHotVector labelNum)

classRatio :: [Label] -> Vec
classRatio labelList = scale (1 / (norm_1 countVec)) $ countVec
    where countVec = labelCount labelList
```

## 4.2 Gini Impurity

$$\mathrm{Gini}(L) = 1 - \sum_{l=0}^{L-1} p_l(L)^2 = 1 - \|\boldsymbol{p}(L)\|_2^2$$

```
gini :: [Label] -> Double
gini labelList = 1.0 - (norm_2 $ classRatio labelList) ^ 2
```

# 5 Search Best Split

## 5.1 Split Data

$$D_l(D, i, v) = \{(\boldsymbol{x}, y) \in D \mid x_i < v\}$$
$$D_r(D, i, v) = \{(\boldsymbol{x}, y) \in D \mid x_i \geq v\}$$

```
splitData :: DataSet -> Literal -> [DataSet]
splitData dataSet (Literal i v) = [lData, rData]
    where
        lData = [(DataPoint x y) | (DataPoint x y) <- dataSet, x !! i <= v]
        rData = [(DataPoint x y) | (DataPoint x y) <- dataSet, x !! i >  v]
```

## 5.2 Score Splitted Data

$$\mathrm{score}(D, i, v) = \frac{|D_l|}{|D|}\mathrm{gini}\left[D_l(D, i, v)\right] + \frac{|D_r|}{|D|}\mathrm{gini}\left[D_r(D, i, v)\right]$$

```
scoreLiteral :: DataSet -> Literal -> Split
scoreLiteral dataSet literal = Split literal score
    where
        score = sum $ map (weightedGini (length dataSet)) $ labelSet
        labelSet = map (map dLabel) $ splitData dataSet literal

weightedGini :: Int -> [Label] -> Double
weightedGini wholeSize labelSet = (gini labelSet) * dblDataSize / dblWholeSize
    where
        dblDataSize     = fromIntegral $ length labelSet
        dblWholeSize    = fromIntegral wholeSize
```

## 5.3 Search Best Split

$$\underset{i,v}{\mathrm{argmin}}\,\mathrm{score}(D, i, v)$$

```
bestSplitAtFeature :: DataSet -> Int -> Split
bestSplitAtFeature dataSet i = myMin splitList
    where
        splitList   = [scoreLiteral dataSet l | l <- literalList]
        literalList = [Literal i (x !! i) | (DataPoint x _) <- dataSet]

bestSplit :: DataSet -> Split
bestSplit dataSet = myMin splitList
    where splitList = [bestSplitAtFeature dataSet f | f <- [0,1..featureNum-1]]
```

# 6 Grow Tree

## 6.1 Grow Tree

```
growTree :: DataSet -> Int -> Int -> String -> Tree
growTree dataSet depth maxDepth nodeId =
    if stopGrowing
        then Leaf (majorLabel dataSet) nodeId
        else Node literal leftTree rightTree nodeId
    where
        literal         = sLiteral $ bestSplit dataSet
        leftTree        = growTree lData (depth + 1) maxDepth (nodeId ++ "l")
        rightTree       = growTree rData (depth + 1) maxDepth (nodeId ++ "r")
        [lData, rData]  = splitData dataSet literal
        stopGrowing =
            depth == maxDepth ||
            gini [y | (DataPoint _ y) <- dataSet] == 0 ||
            length lData == 0 || length rData == 0
```

## 6.2 Stop Growing

$$\text{majorLabel}(D) = \operatorname*{argmax}_{l \in \mathcal{L}} \sum_{(\boldsymbol{x},y) \in D} \mathbb{I}\left[y = l\right]$$

```
majorLabel :: DataSet -> Label
majorLabel dataSet = maxIndex $ labelCount [y | (DataPoint _ y) <- dataSet]
```

## 7 Output Tree in GraphViz

```
labelToStringForGraphViz :: Tree -> String
labelToStringForGraphViz (Leaf l leafId) =
    leafId ++ " [label=\"Class: " ++ (show l) ++ "\"]\n"
labelToStringForGraphViz (Node (Literal i v) left right nodeId) =
    nodeId ++ " [shape=box,label=\"Feature[" ++ (show i) ++ "] < " ++ (show v) ++
        "\"]\n" ++
    labelToStringForGraphViz left ++ labelToStringForGraphViz right

nodeToStringForGraphViz :: Tree -> String
nodeToStringForGraphViz (Leaf _ leafId) = leafId ++ ";\n"
nodeToStringForGraphViz (Node _ left right nodeId) =
    nodeId ++ " -- " ++ nodeToStringForGraphViz left ++
    nodeId ++ " -- " ++ nodeToStringForGraphViz right

treeToStringForGraphViz :: Tree -> String
treeToStringForGraphViz tree =
    "graph Tree {\n" ++ labelToStringForGraphViz tree ++ nodeToStringForGraphViz
        tree ++ "}"
```

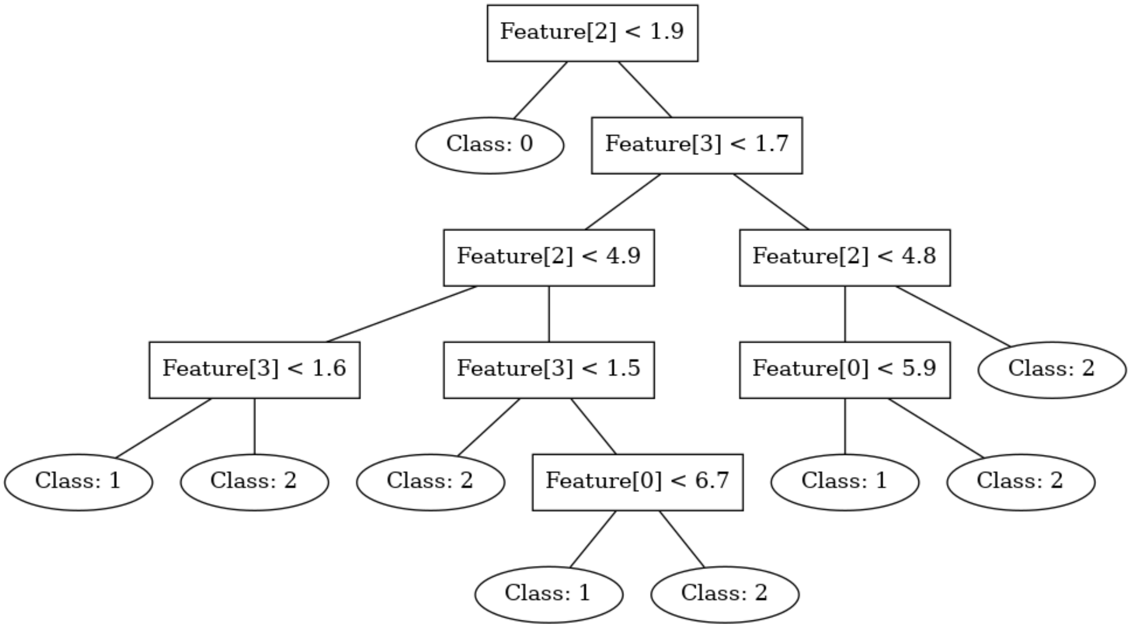

Figure 1: Example of GraphViz output

# 8 Main

```
main :: IO()
main = do
    dataSet <- readDataFromCSV "data/iris/iris.data"
    let tree = growTree dataSet 0 10 "n"
    let treeStr = show tree
    putStrLn treeStr
    writeFile "output/output-tree" treeStr
    writeFile "output/tree.dot" $ treeToStringForGraphViz tree
```

# 9 Other Functions

## 9.1 Algorithm

```
myMin :: [Split] -> Split
myMin splitList = foldr min (Split (Literal 0 0) 2) splitList

oneHotList :: Int -> Int -> [Double]
oneHotList len idx =
    if len == 0
        then []
        else
            if idx == 0
                then 1 : oneHotList (len - 1) (idx - 1)
                else 0 : oneHotList (len - 1) (idx - 1)

oneHotVector :: Int -> Int -> Vec
oneHotVector len idx = vector $ oneHotList len idx
```