CART in Haskell

Genji Ohara

June 17, 2023

1 Preamble

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

import Text.ParserCombinators.Parsec
import Data.CSV
import Data.List

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	$\mathcal{D} = \mathcal{F} \times \mathcal{L}$

```
type Feature    = [Double]
type Label      = Int
type DataSet    = [DataPoint]

data DataPoint = DataPoint {
    dFeature :: Feature,
    dLabel   :: Label
} deriving Show
```

2.2 Constants

```
featureNum :: Int
featureNum = 4

labelNum :: Int
labelNum = 3
```

2.3 Tree Structure

2.3.1 Literal

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

2.3.2 Split

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

instance Eq Split where
    (Split l s) == (Split l' s') = s == s'

instance Ord Split where
    compare (Split l s) (Split l' s') = compare s s'
```

2.3.3 Tree

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

3 Gini Impurity

3.1 Class Ratio

Label Set	$L = \{y \mid (\boldsymbol{x}, y) \in D\}$	
Label Count	$c_l(L) = \sum_{i \in L} \mathbb{I}[i = l],$	$\boldsymbol{c}(L) = \sum_{i \in L} \text{onehot}(i)$
Class Ratio	$p_l(L) = \frac{c_l(L)}{\lvert L \rvert},$	$\boldsymbol{p}(L) = \frac{\boldsymbol{c}(L)}{\lVert \boldsymbol{c}(L) \rVert_1}$

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

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

3.2 Gini Impurity

$$\text{Gini}(L) = 1 - \sum_{l=0}^{L-1} p_l(L)^2 = 1 - \lVert \boldsymbol{p}(L) \rVert_2^2$$

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

4 Search Best Split

4.1 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]
```

4.2 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
```

4.3 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 y) <- dataSet]

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

5 Grow Tree

5.1 Grow Tree

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

5.2 Stop Growing

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

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

6 Output Tree

```
literalToStr :: Literal -> Bool -> String
literalToStr (Literal i v) less =
    "Feature[" ++ (show i) ++ "]" ++ if less then "< " else ">= " ++ (show v)

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

treeToString :: Tree -> Int -> String
treeToString (Leaf label id) depth =
    branchToString depth ++ "class: " ++ (show label) ++ "\n"
treeToString (Node (Literal i v) leftTree rightTree id) depth =
    let
        str1 = branchToString depth ++ "Feature[" ++ (show i) ++ "] "
        str2 = "< " ++ (show v) ++ "\n"
        str3 = treeToString leftTree $ depth + 1
        str4 = ">= " ++ (show v) ++ "\n"
        str5 = treeToString rightTree $ depth + 1
    in str1 ++ str2 ++ str3 ++ str1 ++ str4 ++ str5
```

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
```

7 Output Tree in GraphViz

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

nodeToStringForGraphViz :: Tree -> String
nodeToStringForGraphViz (Leaf label id) = id ++ ";\n"
nodeToStringForGraphViz (Node lLiteral left right id) =
    id ++ " -- " ++ nodeToStringForGraphViz left ++
    id ++ " -- " ++ nodeToStringForGraphViz right

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

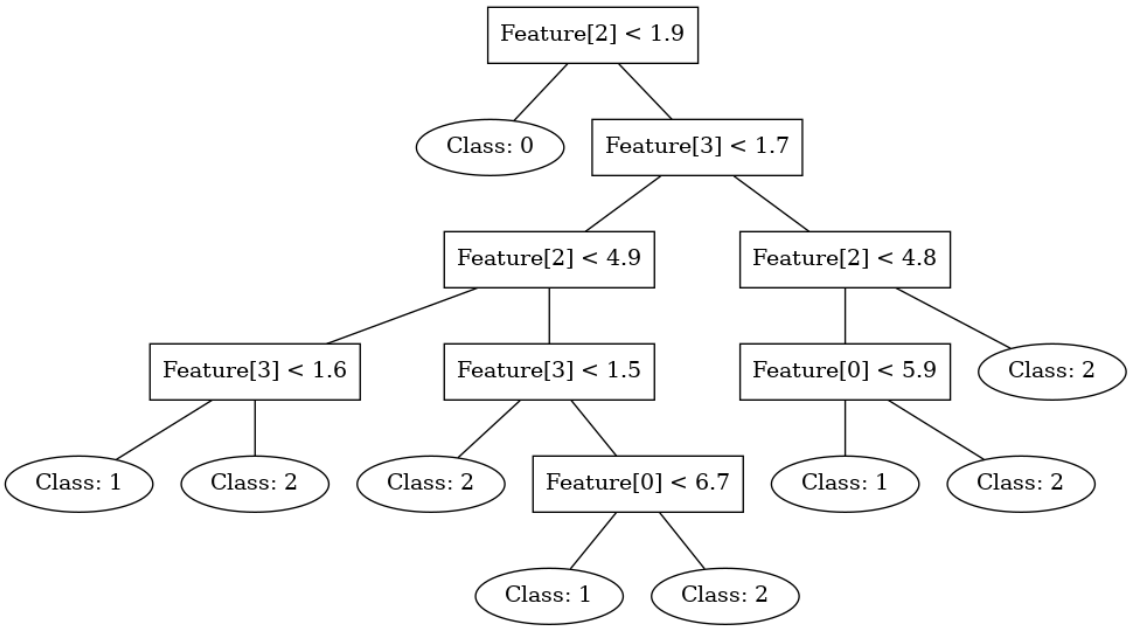

Figure 1: Example of GraphViz output

8 Main

```
main = do
    rawDataSet <- parseFromFile csvFile "data/iris/iris.data"
    let dataSet = either (\x -> []) processData rawDataSet
    let tree = growTree dataSet 0 10 "n"
    let treeStr = treeToString tree 0
    putStrLn treeStr
    writeFile "output/output-tree" treeStr
    writeFile "output/tree.dot" $ treeToStringForGraphViz tree
```

9 Other Functions

9.1 I-O & Data Processing

```
strLabelToIntLabel :: String -> Int
strLabelToIntLabel str
    | str == "Iris-setosa"      = 0
    | str == "Iris-versicolor"  = 1
    | str == "Iris-virginica"   = 2
    | otherwise                 = 3

processDataPoint :: [String] -> DataPoint
processDataPoint strs = DataPoint feature label
    where
        feature = map (read :: String -> Double) $ init strs
        label   = strLabelToIntLabel $ last strs

processData :: [[String]] -> [DataPoint]
processData rawData = map processDataPoint rawData
```

9.2 Algorithm

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

myMax :: [Split] -> Split
myMax splitList = foldr max (Split (Literal 0 0) (-1)) splitList

myMaxIndex :: Ord a => [a] -> Int
myMaxIndex xs = head $ filter ((== maximum xs) . (xs !!)) [0..]

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
```