Haskell-ML

Genji Ohara

March 20, 2024

Chapter 1

Introduction

1.1 Preamble

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

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

1.2 Entry Point

```
-- 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

main :: IO ()
main = do
    cs1 <- readFile "dataset/train_data.dat"
    cs2 <- readFile "dataset/test_data.dat"
    cs3 <- readFile "dataset/train_label.dat"
    cs4 <- readFile "dataset/test_label.dat"

    let batchSize  = 100

    let trainDataList   = map read $ lines cs1
    let testDataList    = map read $ lines cs2
    let trainLabelList  = map read $ lines cs3
    let testLabelList   = map read $ lines cs4
```

```
weight <- flatten <$> randn 1 weight_size

let trainData   = matrix inputSize $ take (batchSize * inputSize) trainDataList
let testData    = matrix inputSize testDataList
let trainLabel  = oneHotMat outputSize $ take batchSize trainLabelList
let testLabel   = oneHotMat outputSize testLabelList

putStr "Now Loading Training Data...\n"
putStr "size of train data  : "
print $ size trainData
putStr "size of train label : "
print $ size trainLabel
putStr "size of test data   : "
print $ size testData
putStr "size of test label  : "
print $ size testLabel

-- putStr "Gradient Check      : "
-- print $ gradientCheck weight x t

let learningRate = 0.1
let iterNum = 100
let newW = learn weight trainData trainLabel learningRate iterNum

print $ testAccuracy newW trainData trainLabel
print $ testAccuracy newW testData testLabel
```

Chapter 2

Decision Tree

2.1 Data Type Definition

2.1.1 Data Space

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

2.1.2 Constants

```
featureNum :: Int
featureNum = 4

labelNum :: Int
labelNum = 3
```

2.1.3 Tree Structure

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)
```

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'
```

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}
```

2.2 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 2.1: 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
```

2.3 Gini Impurity

2.3.1 Class Ratio

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

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

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

2.3.2 Gini Impurity

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

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

2.4 Search Best Split

2.4.1 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]
```

2.4.2 Score Splitted Data

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

```
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
```

2.4.3 Search Best Split

$$\operatorname*{argmin}_{i,v} \text{score}(D, i, v)$$

```
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]]
```

2.5 Grow Tree

2.5.1 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
```

2.5.2 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]
```

2.6 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 2.1: Example of GraphViz output

2.7 Other Functions

2.7.1 Algorithm

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

oneHotList :: Int -> Int -> [R]
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

oneHotMat :: Int -> [Int] -> Mat
oneHotMat len labelList = fromRows $ map (oneHotVector len) labelList
```

Chapter 3

Neural Network

3.1 Constants

```
inputSize   :: Int
hiddenSize  :: Int
outputSize  :: Int
inputSize   = 784
hiddenSize  = 50
outputSize  = 10

w1_start    :: Int
w1_size     :: Int
w2_start    :: Int
w2_size     :: Int
b2_start    :: Int
weight_size :: Int
w1_start    = 0
w1_size     = inputSize * hiddenSize
w2_start    = w1_size + hiddenSize
w2_size     = hiddenSize * outputSize
b2_start    = w2_start + w2_size
weight_size = w1_size + hiddenSize + w2_size + outputSize
```

3.2 Layers

3.2.1 Affine

forward

```
affine :: Mat -> Vec -> Mat -> Mat
affine w b x = x <> w + asRow b

affineDX :: Mat -> Mat -> Mat
affineDX w dout = dout <> (tr w)
```

```
affineDW :: Mat -> Mat -> [R]
affineDW x dout = (matToList $ (tr x) <> dout) ++ (toList $ sum $ toRows dout)
```

3.2.2 Activation Function

ReLU

$$\mathrm{ReLU}(x) = \max(x, 0)$$
$$\mathrm{ReLU}(X) = \begin{bmatrix} \mathrm{ReLU}(x_{11}) & \cdots & \mathrm{ReLU}(x_{1N}) \\ \vdots & \ddots & \vdots \\ \mathrm{ReLU}(x_{N1}) & \cdots & \mathrm{ReLU}(x_{NN}) \end{bmatrix}$$

```
relu :: Mat -> Mat
relu = cmap (max 0)

reluBackward :: Mat -> Mat -> Mat
reluBackward dout x = dout * mask
    where mask = cmap (\_x -> if _x > 0 then 1 else 0) x
```

Sigmoid

$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

```
sigmoid :: R -> R
sigmoid x = 1 / (1 + exp(-x))
```

3.2.3 Cross Entropy Error

$$\mathrm{CEE}(\boldsymbol{y}, \boldsymbol{t}) = -\boldsymbol{t}^T \begin{bmatrix} \ln y_1 \\ \vdots \\ \ln y_D \end{bmatrix}$$
$$\mathrm{CEE}(Y, T) = \sum_{i=1}^{N} \mathrm{CEE}(\boldsymbol{y}_i, \boldsymbol{t}_i)$$

```
sumCrossEntropyError :: [Vec] -> [Vec] -> R
sumCrossEntropyError [] _ = 0
sumCrossEntropyError _ [] = 0
sumCrossEntropyError (y:ys) (t:ts) = -t <.> (cmap log y) + sumCrossEntropyError ys ts

crossEntropyError :: Mat -> Mat -> R
```

```
crossEntropyError y t = sumCrossEntropyError ys ts / batchSize
    where
        ys = toRows y
        ts = toRows t
        batchSize = fromIntegral $ length ys
```

3.2.4 Softmax

Softmax

$$\begin{aligned}
\exp(\boldsymbol{x}) &= \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_N} \end{bmatrix} \\
\operatorname{softmax}(\boldsymbol{x}) &= \frac{\exp(\boldsymbol{x})}{\|\exp(\boldsymbol{x})\|_1} = \frac{\exp(\boldsymbol{x}-\boldsymbol{c})}{\|\exp(\boldsymbol{x}-\boldsymbol{c})\|_1} \\
\operatorname{softmax}(X) &= \begin{bmatrix}\operatorname{softmax}(\boldsymbol{x}_{:1}) & \cdots & \operatorname{softmax}(\boldsymbol{x}_{:N})\end{bmatrix}
\end{aligned}$$

```
softmaxVec :: Vec -> Vec
softmaxVec xVec = scale (1 / norm_1 expVec) expVec
    where
        c       = maxElement xVec
        cVec    = vector $ take (size xVec) [c,c..]
        expVec  = cmap exp $ xVec - cVec

softmax :: Mat -> Mat
softmax = fromRows . (map softmaxVec) . toRows
```

Softmax with Loss

```
softmaxWithLoss :: Mat -> Mat -> R
softmaxWithLoss x t = crossEntropyError (softmax x) t

softmaxWithLossBackward :: Mat -> Mat -> Mat
softmaxWithLossBackward y t = (y - t) / (scalar $ fromIntegral $ rows y)
```

3.2.5 Loss Function

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}; X, T) &= \operatorname{softmaxWithLoss}(\hat{Y}, T) \\
&= \operatorname{CEE}(\operatorname{softmax}(\hat{Y}), T)
\end{aligned}$$

```
loss :: Vec -> Mat -> Mat -> R
loss w x t = softmaxWithLoss (forwardProp w x) t
```

3.2.6 Forward Propagetion

```
--      (softmax)
forwardProp :: Vec -> Mat -> Mat
forwardProp weight x = affine w2 b2 $ relu $ affine w1 b1 x
    where
        w1 = reshape hiddenSize $ subVector w1_start w1_size weight :: Mat
        w2 = reshape outputSize $ subVector w2_start w2_size weight :: Mat
        b1 = subVector w1_size  hiddenSize weight
        b2 = subVector b2_start outputSize weight
```

3.2.7 Prediction

```
predict :: Vec -> Mat -> Mat
predict w x = oneHotMat outputSize $ map maxIndex $ toRows $ forwardProp w x
```

3.2.8 Gradient

```
numericalGradientList :: Int -> (Vec -> R) -> Vec -> [R]
numericalGradientList idx f x =
    if idx == size x
    then []
    else
    let h  = 1e-4
        dx = cmap (* h) $ oneHotVector (size x) idx
        x1 = x + dx
        x2 = x - dx
    in (f(x1) - f(x2)) / (2 * h) : numericalGradientList (idx + 1) f x

numericalGradient :: (Vec -> R) -> Vec -> Vec
numericalGradient f = vector . (numericalGradientList 0 f)

matToList :: Mat -> [R]
matToList = concat . toLists

gradient :: Vec -> Mat -> Mat -> Vec
gradient weight x t =

    let w1 = reshape hiddenSize $ subVector w1_start w1_size weight :: Mat
        w2 = reshape outputSize $ subVector w2_start w2_size weight :: Mat
        b1 = subVector w1_size  hiddenSize weight
        b2 = subVector b2_start outputSize weight

        -- forward propagation
        a1 = affine w1 b1 x
        y1 = relu a1
        y2 = softmax $ affine w2 b2 y1
```

```
        -- backward propagation
        da2 = softmaxWithLossBackward y2 t
        dx2 = affineDX w2 da2
        dw2 = affineDW y1 da2
        da1 = reluBackward dx2 a1
        dw1 = affineDW x da1

    in fromList $ dw1 ++ dw2
  --

gradientCheck :: Vec -> Mat -> Mat -> R
gradientCheck w x t =
    let num_grad    = numericalGradient (\_w -> loss _w x t) w
        grad        = gradient w x t
        err_sum     = sum $ map abs $ toList $ num_grad - grad
    in err_sum / (fromIntegral $ length $ toList grad)
```

3.2.9 Learning

```
learn :: Vec -> Mat -> Mat -> R -> R -> Vec
learn weight x t learningRate iterNum =
    if iterNum == 0
    then weight
    else learn new_w x t learningRate (iterNum - 1)
    where
    new_w = weight - (cmap (* learningRate) $ gradient weight x t)

testAccuracy :: Vec -> Mat -> Mat -> R
testAccuracy w x t = scoreSum / (fromIntegral $ rows x)
    where scoreSum = sumElements $ takeDiag $ (predict w x) <> (tr t)
```