Haskell-ML

Genji Ohara

March 23, 2024

Contents

Chapter 1

Introduction

1.1 About This Book

This book is a collection of Haskell code for machine learning. This PDF file is generated from haskell-ml.lhs written in Literate Haskell format. You can compile it as both Haskell and LaTeX source code. I write this book to learn Haskell and machine learning and hope it will be helpful for those who have the same interest.

1.2 Prerequisites

We use the following libraries:

- `Prelude` for basic functions
- `Numeric.LinearAlgebra` for matrix operations
- `Data.CSV` for reading CSV files
- `Text.ParserCombinators.Parsec` for parsing CSV files
- `System.Random` for random number generation
- `Data.List` for list operations

```haskell
import Prelude hiding ((<>))
import Numeric.LinearAlgebra
import Data.CSV
import Text.ParserCombinators.Parsec
import System.Random
import List.Shuffle
import Data.List
import Text.Printf
```

We use the following type aliases:

- `R` for `Double`
- `Vec` for `Vector R`
- `Mat` for `Matrix R`

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

We define the some spaces as follows:

$$\text{Feature Space} \qquad \mathcal{F} = \mathbb{R}^D \tag{1.1}$$

$$\text{Label Space} \qquad \mathcal{L} = \{0, 1, \ldots, L-1\} \tag{1.2}$$

$$\text{Data Space} \qquad \mathcal{D} = \mathcal{F} \times \mathcal{L} \tag{1.3}$$

```
type Feature    = [Double]
type Label      = Int
data DataPoint = DataPoint {
    dFeature :: Feature,
    dLabel   :: Label
} deriving Show
data RegDataPoint = RegDataPoint {
    rdFeature :: Feature,
    rdLabel   :: Double
} deriving Show
```

You can test all methods in this book by compiling haskell-ml.lhs as a Haskell source code.

```
main :: IO()
main = do
    testCls
    testReg
```

1.3 Data Processing

1.3.1 Read Data

We need to read external datasets for input to models.

```
type DataSet    = [DataPoint]
type RegDataSet = [RegDataPoint]
```

```
readClsDataFromCSV :: String -> IO DataSet
readClsDataFromCSV fileName = do
    rawData <- parseFromFile csvFile fileName
    return $ either (const []) processClsData rawData

readRegDataFromCSV :: String -> IO RegDataSet
readRegDataFromCSV fileName = do
    rawData <- parseFromFile csvFile fileName
    return $ either (const []) processRegData rawData
```

1.3.2 Process Data

We need following steps to process data:

$$\begin{tikzcd}
& \texttt{rawData} \arrow[dl, "\texttt{init}"'] \arrow[dr, "\texttt{last.transpose}"] & \\
\texttt{rawFeat} \arrow[d, "\texttt{map (map read)}"'] & & \texttt{rawLabs} \arrow[d, "\texttt{strLabelToIntLabel}"] \\
\texttt{feats} \arrow[dr, "\texttt{concatDataPoint}"] & & \texttt{labs} \arrow[dl] \\
& \text{Processed Data} &
\end{tikzcd}$$

```
processClsData :: [[String]] -> [DataPoint]
processClsData rawData = concatClsDataPoint feats labs
    where
        rawLabs = (last . transpose) rawData
        feats   = map (map (read :: String -> Double) . init) rawData
        labs    = strLabelToIntLabel rawLabs

processRegData :: [[String]] -> [RegDataPoint]
processRegData rawData = concatRegDataPoint feats labs
    where
        rawLabs = (last . transpose) rawData
        feats   = map (map (read :: String -> R) . init) rawData
        labs    = map (read :: String -> R) rawLabs

strLabelToIntLabel :: [String] -> [Int]
strLabelToIntLabel strLabels = map (maybeToInt . labelToIndex) strLabels
    where
        labelToIndex l = elemIndex l $ nub strLabels
        maybeToInt Nothing = 0
        maybeToInt (Just a) = a
```

```haskell
concatClsDataPoint :: [[Double]] -> [Int] -> [DataPoint]
concatClsDataPoint (f:fs) (l:ls) = DataPoint f l : concatClsDataPoint fs ls
concatClsDataPoint [] _ = []
concatClsDataPoint _ [] = []

concatRegDataPoint :: [[Double]] -> [Double] -> [RegDataPoint]
concatRegDataPoint (f:fs) (l:ls) = RegDataPoint f l : concatRegDataPoint fs ls
concatRegDataPoint [] _ = []
concatRegDataPoint _ [] = []
```

1.3.3 Split Data

We need to split the dataset into training and test datasets.

```haskell
splitDataset :: [a] -> R -> StdGen -> ([a], [a])
splitDataset dataSet rate gen = (trainData, testData)
    where
        shuffledData = fst $ shuffle dataSet gen
        trainData = take (round $ rate * fromIntegral (length shuffledData))
            shuffledData
        testData  = drop (round $ rate * fromIntegral (length shuffledData))
            shuffledData
```

Chapter 2

Linear Model

2.1 Overview

In this chapter, we introduce linear models below:

- Linear Regression
- Logistic Regression
- Support Vector Machine (SVM)

2.2 Linear Regression

Linear regression is a very simple regressor.

2.2.1 Setting

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$, where $\boldsymbol{x}_i \in \mathbb{R}^D$ is a feature vector and $y_i \in \{0, 1\}$ is a label:

$$\boldsymbol{X} \triangleq \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \quad \boldsymbol{y} \triangleq \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.1}$$

2.2.2 Model

We get the estimated label $\hat{y}$ from the feature vector $\boldsymbol{x}$ as follows:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{2.2}$$

We transform eq. (2.2) by adding a bias term:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + w_0 = \begin{bmatrix} w_0 & \boldsymbol{w}^T \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}}. \tag{2.3}$$

```
predictLinReg :: Vec -> Vec -> R
predictLinReg tw x = tw <.> vector (1.0 : toList x)

predictLinRegMat :: Vec -> Mat -> Vec
predictLinRegMat tw x = fromList $ map (predictLinReg tw) $ toRows x
```

2.2.3 Problem

We want to find the weight $\tilde{\boldsymbol{w}}$ that minimizes the objective:

$$E(\tilde{\boldsymbol{w}}) = \|\boldsymbol{y} - \tilde{\boldsymbol{X}}\tilde{\boldsymbol{w}}\|^2 + \lambda \|\tilde{\boldsymbol{w}}\|^2. \tag{2.4}$$

where

$$\tilde{\boldsymbol{X}} \triangleq \begin{bmatrix} \tilde{\boldsymbol{x}}_1^T \\ \tilde{\boldsymbol{x}}_2^T \\ \vdots \\ \tilde{\boldsymbol{x}}_N^T \end{bmatrix} = \begin{bmatrix} 1 & \\ 1 & \\ \vdots & \boldsymbol{X} \\ 1 & \end{bmatrix} \tag{2.5}$$

```
addBias :: Mat -> Mat
addBias x = fromColumns $ bias : toColumns x
    where bias = vector $ take (rows x) [1,1..]
```

2.2.4 Fitting

Gradient of the objective eq. (2.4) is

$$\nabla E(\tilde{\boldsymbol{w}}) = 2 \left[\left(\tilde{\boldsymbol{X}}^T \tilde{\boldsymbol{X}} + \lambda I \right) \tilde{\boldsymbol{w}} - \tilde{\boldsymbol{X}}^T \boldsymbol{y} \right]. \tag{2.6}$$

Therefore

$$\operatorname*{argmin}_{\tilde{\boldsymbol{w}}} E(\tilde{\boldsymbol{w}}) = \left(\tilde{\boldsymbol{X}}^T \tilde{\boldsymbol{X}} + \lambda I \right)^{-1} \tilde{\boldsymbol{X}}^T \boldsymbol{y} \tag{2.7}$$

```
fitLinReg :: Mat -> Vec -> R -> Vec
fitLinReg x y lambda = inv a #> (tr x_til #> y)
    where
        a = tr x_til <> x_til + scale lambda (ident $ cols x_til)
        x_til = addBias x
```

2.3 Logistic Regression

2.3.1 Model

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x}), \tag{2.8}$$

where σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{2.9}$$

```
predictProbOneLogReg :: Vec -> Vec -> R
predictProbOneLogReg w x = sigmoid $ w <.> x

predictProbLogReg :: Vec -> Mat -> Vec
predictProbLogReg w x = fromList $ map (predictProbOneLogReg w) $ toRows x

predictOneLogReg :: Vec -> Vec -> Int
predictOneLogReg w x = if predictProbOneLogReg w x > 0.5 then 1 else 0

predictLogReg :: Vec -> Mat -> Vector Int
predictLogReg w x = fromList $ map (predictOneLogReg w) $ toRows x

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

2.3.2 Problem

We minimize the objective:

$$E(\boldsymbol{w}) = -\sum_{i=1}^{N} \left[t_i \ln \hat{y}_i + (1 - t_i) \ln(1 - \hat{y}_i)\right] + \frac{\lambda}{2} \|\boldsymbol{w}\|^2 \tag{2.10}$$

Gradient:

$$\nabla E(\boldsymbol{w}) = \boldsymbol{X}^T(\hat{\boldsymbol{y}} - \boldsymbol{t}) + \lambda \boldsymbol{w} \tag{2.11}$$

```
gradientLogReg :: Vec -> Mat -> Vec -> R -> Vec
gradientLogReg w x t lambda = tr x #> (ys - t) + scale lambda w
    where
        ys = predictProbLogReg w x
```

2.3.3 Fitting

Stochastic Gradient Descent

We update the weight as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla E(\boldsymbol{w}) \tag{2.12}$$

By iterating eq. (2.12), we can minimize the objective eq. (2.10).

```
sgd :: Vec -> Mat -> Vec -> R -> R -> Vec
sgd weight x t learningRate iterNum =
    if iterNum == 0
    then weight
    else sgd new_w x t learningRate (iterNum - 1)
    where
    new_w = weight - cmap (* learningRate) (gradientLogReg weight x t 0.1)
```

2.4 Support Vector Machine (SVM)

2.4.1 Model

$$\hat{y} = \mathrm{sign}(\boldsymbol{w}^T \boldsymbol{x} + w_0) \tag{2.13}$$

where

$$\mathrm{sign}(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}. \tag{2.14}$$

```
predictOneSVM :: Vec -> Vec -> Int
predictOneSVM w x = if w <.> x > 0 then 1 else 0

predictSVM :: Vec -> Mat -> Vector Int
predictSVM w x = fromList $ map (predictOneSVM w) $ toRows x
```

2.4.2 Problem

We minimize the objective:

$$E(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N} \xi_i \tag{2.15}$$

subject to

$$\xi_i \geq 0, \quad i = 1, \ldots, N, \tag{2.16}$$

$$t_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) \geq 1 - \xi_i, \quad i = 1, \ldots, N. \tag{2.17}$$

2.4.3 Fitting

We can solve the problem by using the Lagrange dual problem.

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j \boldsymbol{x}_i^T \boldsymbol{x}_j \tag{2.18}$$

subject to

$$0 \leq \alpha_i \leq C, \quad i = 1, \ldots, N, \tag{2.19}$$

$$\sum_{i=1}^{N} \alpha_i t_i = 0. \tag{2.20}$$

```haskell
fitSVM :: Mat -> Vec -> R -> R -> Vec
fitSVM x t c lambda = inv a #> (tr x #> t)
    where
        a = tr x <> x + scale lambda (ident $ cols x)
```

2.4.4 Kernel Trick

We can use the kernel trick to solve the problem in a high-dimensional space.

$$\hat{y} = \mathrm{sign}\left(\sum_{i=1}^{N} \alpha_i t_i K(\boldsymbol{x}_i, \boldsymbol{x}) + w_0 \right), \tag{2.21}$$

where $K(\boldsymbol{x}, \boldsymbol{x}')$ is kernel function.

```haskell
fitSVMKernel :: Mat -> Vec -> R -> R -> Vec
fitSVMKernel x t c lambda = inv a #> t
    where
        a = tr k <> k + scale lambda (ident $ cols k)
        k = kernel x
```

Kernel functions are as follows:

- Linear kernel: $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$
- Polynomial kernel: $K(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + 1)^d$
- Gaussian kernel: $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2\sigma^2}\right)$

When we use Linear kernel, it is the equivalent to the original SVM.

```
polynomialKernel :: Int -> Vec -> Vec -> R
polynomialKernel d x x' = (1 + (x <.> x')) ^ d

gaussianKernel :: R -> Vec -> Vec -> R
gaussianKernel sigma x x' = exp $ (-1) * (norm_2 $ x - x') ^ 2 / (2 * sigma ^ 2)
```

Chapter 3

Tree Model

3.1 Decision Tree

3.1.1 Model

Constants

```
featureNum :: Int
featureNum = 4

labelNum :: Int
labelNum = 3
```

Literal

```
data Literal = Literal Int 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}
```

Prediction

```
predictTree :: Tree -> Feature -> Int
predictTree (Leaf l _) _ = l
predictTree (Node (Literal i v) left right _) x =
    if x !! i <= v
    then predictTree left x
    else predictTree right x

predictDT :: Tree -> Mat -> Vector Int
predictDT tree x = fromList $ map ((predictTree tree) . toList) $ toRows x
```

3.1.2 Problem

Class Ratio

$$\text{Label Set} \qquad L = \{y \mid (\boldsymbol{x}, y) \in D\} \tag{3.1}$$

$$\text{Label Count} \qquad c_l(L) = \sum_{i \in L} \mathbb{I}[i = l], \qquad \boldsymbol{c}(L) = \sum_{i \in L} \text{onehot}(i) \tag{3.2}$$

$$\text{Class Ratio} \qquad p_l(L) = \frac{c_l(L)}{|L|}, \qquad \boldsymbol{p}(L) = \frac{\boldsymbol{c}(L)}{\|\boldsymbol{c}(L)\|_1} \tag{3.3}$$

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

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

Gini Impurity

$$\mathrm{Gini}(L) = 1 - \sum_{l=0}^{L-1} p_l(L)^2 = 1 - \|\boldsymbol{p}(L)\|_2^2 \tag{3.4}$$

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

3.1.3 Search Best Split

Split Data

$$D_l(D,i,v) = \{(\boldsymbol{x},y) \in D \mid x_i < v\} \tag{3.5}$$
$$D_r(D,i,v) = \{(\boldsymbol{x},y) \in D \mid x_i \geq v\} \tag{3.6}$$

```
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]
```

Score Splitted Data

$$\mathrm{score}(D,i,v) = \frac{|D_l|}{|D|}\mathrm{gini}\left[D_l(D,i,v)\right] + \frac{|D_r|}{|D|}\mathrm{gini}\left[D_r(D,i,v)\right] \tag{3.7}$$

```
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
```

Search Best Split

$$\operatorname*{argmin}_{i,v} \operatorname{score}(D, i, v) \tag{3.8}$$

```
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]]
```

3.1.4 Grow Tree

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
```

Stop Growing

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

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

3.1.5 Output Tree

For CLI

```haskell
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 3.1: 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
```

For 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
        ++ "}"
```

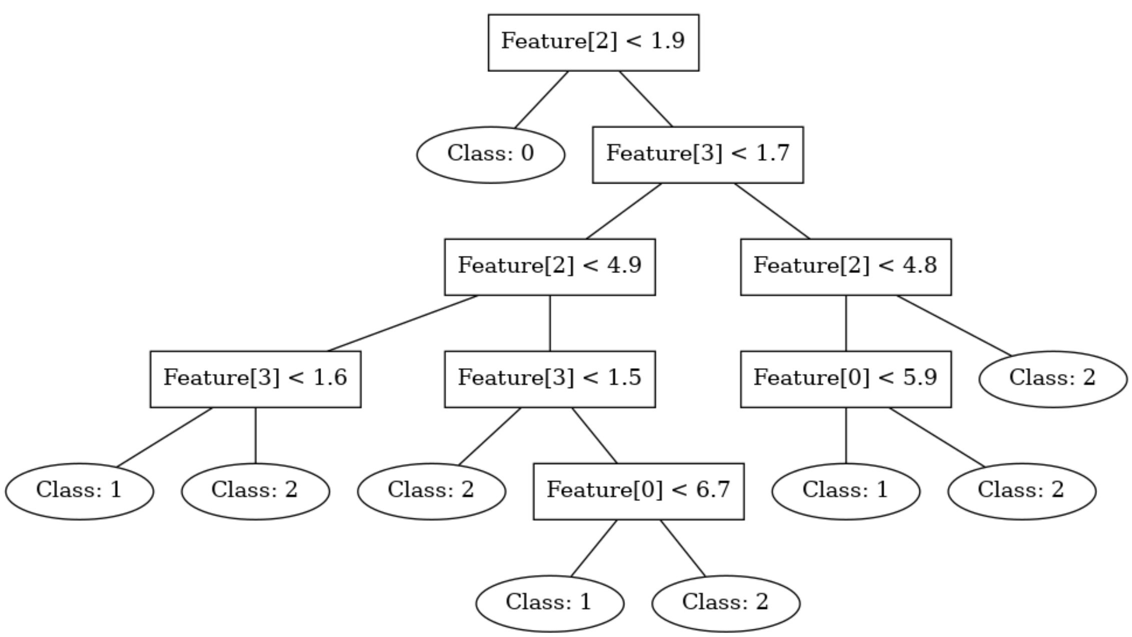

Figure 3.1: Example of GraphViz output

3.1.6 Other Functions

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 4

Neural Network

4.1 Constants

```
inputSize   :: Int
hiddenSize  :: Int
outputSize  :: Int
inputSize   = 4
hiddenSize  = 8
outputSize  = 3

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
```

4.2 Layers

4.2.1 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)
```

4.2.2 Activation Function

ReLU

$$\mathrm{ReLU}(x) = \max(x, 0) \tag{4.1}$$

$$\mathrm{ReLU}(X) = \begin{bmatrix} \mathrm{ReLU}(x_{11}) & \cdots & \mathrm{ReLU}(x_{1N}) \\ \vdots & \ddots & \vdots \\ \mathrm{ReLU}(x_{N1}) & \cdots & \mathrm{ReLU}(x_{NN}) \end{bmatrix} \tag{4.2}$$

```
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

See eq. (2.9).

4.2.3 Cross Entropy Error

$$\mathrm{CEE}(\boldsymbol{y}, \boldsymbol{t}) = -\boldsymbol{t}^T \begin{bmatrix} \ln y_1 \\ \vdots \\ \ln y_D \end{bmatrix} \tag{4.3}$$

$$\mathrm{CEE}(Y, T) = \sum_{i=1}^{N} \mathrm{CEE}(\boldsymbol{y}_i, \boldsymbol{t}_i) \tag{4.4}$$

```
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
```

4.2.4 Softmax

Softmax

$$\exp(\boldsymbol{x}) = \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_N} \end{bmatrix} \tag{4.5}$$

$$\operatorname{softmax}(\boldsymbol{x}) = \frac{\exp(\boldsymbol{x})}{\|\exp(\boldsymbol{x})\|_1} = \frac{\exp(\boldsymbol{x}-\boldsymbol{c})}{\|\exp(\boldsymbol{x}-\boldsymbol{c})\|_1} \tag{4.6}$$

$$\operatorname{softmax}(X) = \begin{bmatrix}\operatorname{softmax}(\boldsymbol{x}_{:1}) & \cdots & \operatorname{softmax}(\boldsymbol{x}_{:N})\end{bmatrix} \tag{4.7}$$

```
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)
```

4.2.5 Loss Function

$$\mathcal{L}(\boldsymbol{w}; X, T) = \text{softmaxWithLoss}(\hat{Y}, T) \tag{4.8}$$
$$= \text{CEE}(\text{softmax}(\hat{Y}), T) \tag{4.9}$$

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

4.2.6 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
```

4.2.7 Prediction

```
predictNN :: Vec -> Mat -> Vector Int
predictNN w x = fromList $ map maxIndex $ toRows $ forwardProp w x
```

4.2.8 Gradient

Numerical Gradient

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i - h, \dots, x_N)}{2h} \tag{4.10}$$

```
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)
```

Backward Propagation

```
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
  --
```

Check Backward Propagation

```
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)
```

4.2.9 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)
```

Chapter 5

Comparison of Models

5.1 Classification

```
testAccuracy :: (Mat -> Vector Int) -> DataSet -> R
testAccuracy predict testData = 1 - norm_2 d_y / (fromIntegral $ rows x)
    where
        x = fromRows $ map (vector . dFeature) testData :: Mat
        y = fromList $ map dLabel testData :: Vector Int
        y_pred = predict x :: Vector Int
        r_y = fromList $ map fromIntegral $ toList $ y :: Vec
        r_y_pred = fromList $ map fromIntegral $ toList $ y_pred :: Vec
        d_y = r_y - r_y_pred


testCls :: IO()
testCls = do
    putStrLn "Classification (Accuracy)"

    -- Data Processing
    dataSet <- readClsDataFromCSV "data/iris/iris.data"
    gen <- getStdGen
    let splittedData = splitDataset dataSet 0.8 gen
    let trainData = fst splittedData
    let testData = snd splittedData
    let x = fromRows $ map (vector . dFeature) trainData
    let y = fromList $ map (fromIntegral . dLabel) trainData

    -- Logistic Regression
    let wLogReg = sgd (vector $ take (cols x) [0,0..]) x y 0.01 1000
    let accLogReg = testAccuracy (predictLogReg wLogReg) testData

    -- SVM
```

```
    let wSVM = fitSVM x y 0.1 1000
    let accSVM = testAccuracy (predictSVM wSVM) testData

    -- Decision Tree
    let tree = growTree trainData 0 10 "0"
    let accDT = testAccuracy (predictDT tree) testData

    -- Neural Network
    let yOnehot = oneHotMat outputSize $ map dLabel trainData
    let w0 = vector $ take weight_size [1,1..]
    let w = learn w0 x yOnehot 0.1 1000
    let accNN = testAccuracy (predictNN w) testData

    putStrLn $ printf "Logistic Regression: %.3f" accLogReg
    putStrLn $ printf "SVM                : %.3f" accSVM
    putStrLn $ printf "Decision Tree      : %.3f" accDT
    putStrLn $ printf "Neural Network     : %.3f" accNN
    putStrLn ""
```

5.2 Regression

```
testMSE :: (Mat -> Vec) -> RegDataSet -> R
testMSE predict testData = (d_y <.> d_y) / (fromIntegral $ rows x)
    where
        x = fromRows $ map (vector . rdFeature) testData
        y = vector $ map rdLabel testData
        d_y = y - (predict x)

testReg :: IO()
testReg = do
    putStrLn "Regression (MSE: Mean Squared Error)"

    -- Data Processing
    dataSet <- readRegDataFromCSV "data/housing.csv"
    gen <- getStdGen
    let splittedData = splitDataset dataSet 0.8 gen
    let trainData = fst splittedData
    let testData = snd splittedData
    let x = fromRows $ map (vector . rdFeature) trainData
    let y = fromList $ map rdLabel trainData

    -- Linear Regression
    let w = fitLinReg x y 0.1
```

```
let mseLinReg = testMSE (predictLinRegMat w) testData

putStrLn $ printf "Linear Regression  : %.3f" mseLinReg
```