

Haskell-ML

Genji Ohara

June 14, 2025

Contents

1	Introduction	3
1.1	About This Book	3
1.2	Prerequisites	3
1.3	Data Processing	4
2	Linear Model	7
2.1	Overview	7
2.2	Linear Regression	7
2.3	Logistic Regression	9
2.4	Support Vector Machine (SVM)	10
3	Tree Model	12
3.1	Decision Tree	12
4	Neural Network	19
4.1	Constants	19
4.2	Layers	19
5	Comparison of Models	25
5.1	Classification	25
5.2	Regression	26

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 $\text{\LaTeX}$ 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
- `Data.CSV` and `Text.ParserCombinators.Parsec` for reading CSV files
- `Numeric.LinearAlgebra` for matrix operations
- `System.Random` and `List.Shuffle` for randomization
- `Text.Printf` for formatted output

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

We use the following type aliases:

- `Vec` for `Vector R`

- Mat for Matrix R

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

We define the some spaces as follows:

$$\text{Feature Space} \quad \mathcal{F} = \mathbb{R}^D \quad (1.1)$$

$$\text{Label Space} \quad \mathcal{L} = \{0, 1, \dots, L - 1\} \quad (1.2)$$

$$\text{Data Space} \quad \mathcal{D} = \mathcal{F} \times \mathcal{L} \quad (1.3)$$

```
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
```

```

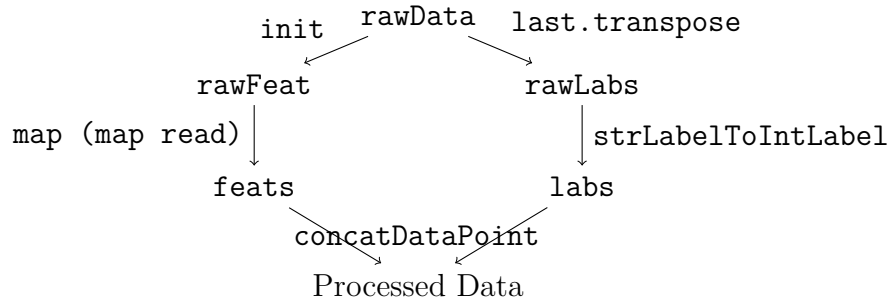
return $ either (const []) processClsData rawData

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:



```

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

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

strLabelToIntLabel :: [String] -> [Int]
strLabelToIntLabel strLabels = map (flip elemIndex labelSet) strLabels
    where
        labelSet = foldl (\xs x -> if x `elem` xs then xs else (x:xs)) [] strLabels
        elemIndex t xs = elemIndex' t xs 0
        where
            elemIndex' _ [] i = i
            elemIndex' t' (x':xs') i
                | t' == x' = i
                | otherwise = elemIndex' t' xs' (i + 1)

```

```

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.

```

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

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

2.2.2 Model

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

$$\hat{y} = \mathbf{w}^T \mathbf{x} + w_0 \quad (2.2)$$

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

$$\hat{y} = \mathbf{w}^T \mathbf{x} + w_0 = \begin{bmatrix} w_0 & \mathbf{w}^T \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}. \quad (2.3)$$

```

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 $\tilde{\mathbf{w}}$ that minimizes the objective:

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

where

$$\tilde{\mathbf{X}} \triangleq \begin{bmatrix} \tilde{\mathbf{x}}_1^T \\ \tilde{\mathbf{x}}_2^T \\ \vdots \\ \tilde{\mathbf{x}}_N^T \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & \ddots \\ & & & 1 \end{bmatrix} \mathbf{X} \quad (2.5)$$

```

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

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

Therefore

$$\underset{\tilde{\mathbf{w}}}{\operatorname{argmin}} E(\tilde{\mathbf{w}}) = \left(\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} + \lambda I \right)^{-1} \tilde{\mathbf{X}}^T \mathbf{y} \quad (2.7)$$

```

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

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

where σ is the sigmoid function:

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

```

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:

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

Gradient:

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

```

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:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E(\mathbf{w}) \quad (2.12)$$

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

$$\hat{y} = \text{sign}(\mathbf{w}^T \mathbf{x} + w_0) \quad (2.13)$$

where

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

```
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:

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

subject to

$$\xi_i \geq 0, \quad i = 1, \dots, N, \quad (2.16)$$

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

2.4.3 Fitting

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

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

subject to

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, N, \quad (2.19)$$

$$\sum_{i=1}^N \alpha_i t_i = 0. \quad (2.20)$$

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

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

$$\text{Label Set} \quad L = \{y \mid (\mathbf{x}, y) \in D\} \quad (3.1)$$

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

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

```
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

$$\text{Gini}(L) = 1 - \sum_{l=0}^{L-1} p_l(L)^2 = 1 - \|\mathbf{p}(L)\|_2^2 \quad (3.4)$$

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

3.1.3 Search Best Split

Split Data

$$D_l(D, i, v) = \{(\mathbf{x}, y) \in D \mid x_i < v\} \quad (3.5)$$

$$D_r(D, i, v) = \{(\mathbf{x}, y) \in D \mid x_i \geq v\} \quad (3.6)$$

```
splitData :: DataSet -> Literal -> (DataSet, 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

$$\text{score}(D, i, v) = \frac{|D_l|}{|D|} \text{gini}[D_l(D, i, v)] + \frac{|D_r|}{|D|} \text{gini}[D_r(D, i, v)] \quad (3.7)$$

```
scoreLiteral :: DataSet -> Literal -> Split
scoreLiteral dataSet literal = Split literal score
  where
    score = sum $ both (weightedGini (length dataSet)) labelSet
    labelSet = both (map dLabel) $ splitData dataSet literal
    both :: (a -> b) -> (a,a) -> (b,b)
    both f (x,y) = (f x, f y)

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

Search Best Split

$$\underset{i,v}{\operatorname{argmin}} \operatorname{score}(D, i, v) \quad (3.8)$$

```

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 ||
      null lData || null rData

```

Stop Growing

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

```

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

```

3.1.5 Output Tree

For CLI

```
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
--   ++ "}"
```

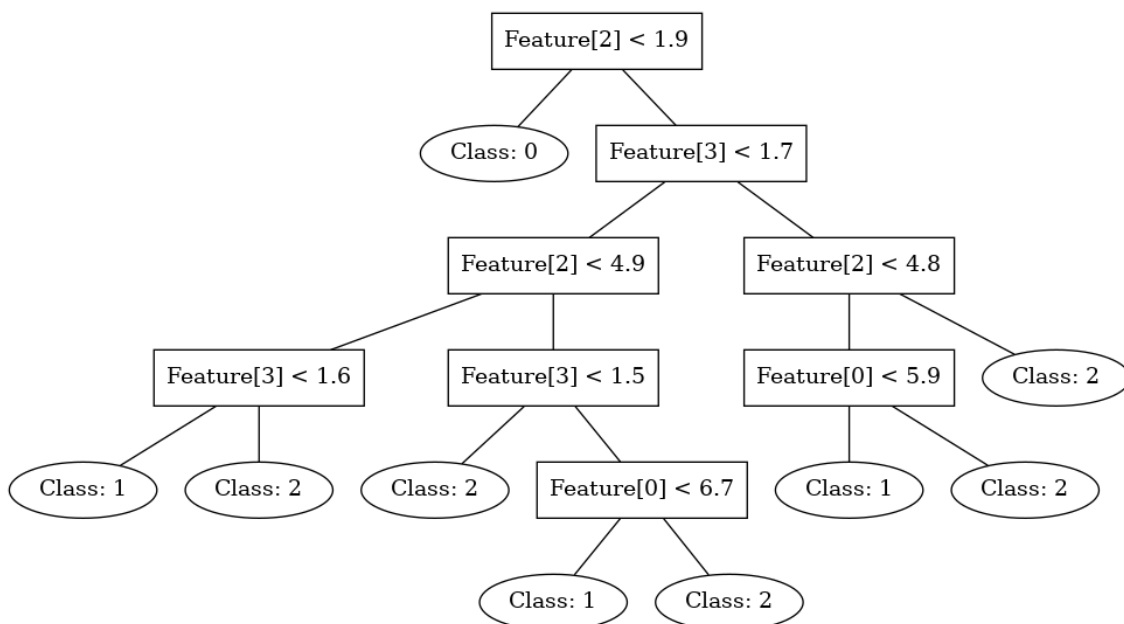


Figure 3.1: Example of GraphViz output

3.1.6 Other Functions

Algorithm

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

oneHotList :: Int -> Int -> [R]
oneHotList len idx
  | len == 0 = []
  | idx == 0 = 1 : oneHotList (len - 1) (idx - 1)
  | otherwise = 0 : oneHotList (len - 1) (idx - 1)

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

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

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

$$\text{ReLU}(x) = \max(x, 0) \quad (4.1)$$

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

```

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

$$\text{CEE}(\mathbf{y}, \mathbf{t}) = -\mathbf{t}^T \begin{bmatrix} \ln y_1 \\ \vdots \\ \ln y_D \end{bmatrix} \quad (4.3)$$

$$\text{CEE}(Y, T) = \sum_{i=1}^N \text{CEE}(\mathbf{y}_i, \mathbf{t}_i) \quad (4.4)$$

```

-- 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

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

$$\text{softmax}(\mathbf{x}) = \frac{\exp(\mathbf{x})}{\|\exp(\mathbf{x})\|_1} = \frac{\exp(\mathbf{x} - \mathbf{c})}{\|\exp(\mathbf{x} - \mathbf{c})\|_1} \quad (4.6)$$

$$\text{softmax}(X) = [\text{softmax}(\mathbf{x}_{:1}) \quad \cdots \quad \text{softmax}(\mathbf{x}_{:N})] \quad (4.7)$$

```

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 = crossEntropyError (softmax x)

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

```

4.2.5 Loss Function

$$\mathcal{L}(\mathbf{w}; X, T) = \text{softmaxWithLoss}(\hat{Y}, T) \quad (4.8)$$

$$= \text{CEE}(\text{softmax}(\hat{Y}), T) \quad (4.9)$$

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

4.2.6 Forward Propagation

```
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

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

```
-- 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_1(Accuracy)"

  -- Data Processing
  dataSet <- readClsDataFromCSV "app/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 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 weightSize [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 "app/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
```
