**Binary and Multiclass Classification using Neural Networks**

by

Akeem Ajede

A report submitted in partial fulfillment of the

requirements for a Masters in

Data Science

Auburn, Alabama

Spring, 2020



Supervisor

Bo Liu, PhD

Department of Computer Science & Engineering

**ABSTRACT**

This analytical study was executed using the R programming language. The dataset used is the famous MNIST dataset [1]. The dataset is a multiclass data from which the training and testing data were sampled. Neural Network with backpropagation, which is an advanced Machine Learning algorithm was implemented as a binary and multiclass classifier. Then, a decision tree algorithm was executed. Last, the accuracy and confusion matrix of each model was reported.

Keyword: Neural Network, Algorithm, Machine Learning

**INTRODUCTION**

Machine Learning (ML) is a branch of data science that requires minimal amount of human intervention to find subtle patterns in large databases and make analytical decisions. It can also be considered as a branch of artificial intelligence based on the idea that machines can learn from data beyond the normal capability of human.

There are several algorithms used in ML, depending on the specific task to be accomplished. In this project, Neural Network (NN) with backpropagation, the decision tree, and the polynomial kernel perceptron were implemented using the MNIST dataset.

A NN is an algorithm, modeled loosely after the human brain, which is designed to recognize patterns in a dataset.Backpropagation in NN is the essence of neural net training. It is a method of improving the weights of a NN based on the error rate (i.e. loss) obtained in the previous epoch or iteration.

Vanilla decision trees are built using a recursive partitioning, which is commonly referred to as split and conquer because the algorithm splits the data into subsets, which are further split until the algorithm deduce that the data within the subsets are sufficiently homogenous, or until the specified stopping criterion is satisfied [2].

**PURPOSE OF STUDY & SCOPE**

The aim of this study is to implement binary and multiclass classification using NN and the decision tree. The resulting models are evaluated using the confusion matrix. The programming language used is R.

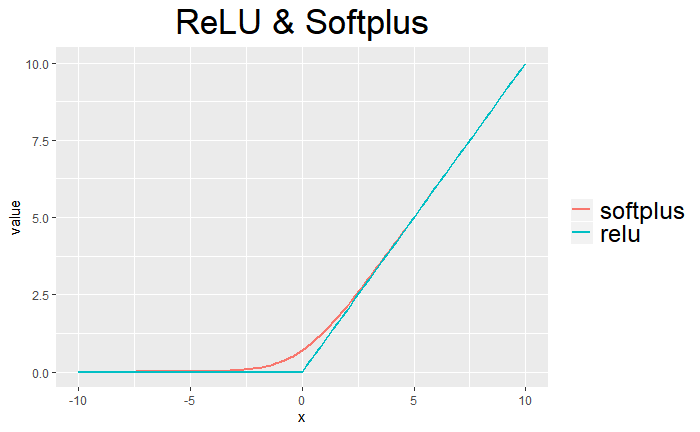
**DATA DESCRIPTION**

The raw MNIST dataset comprises of images of handwritten digits, and our labels are a single digit that indicate the number written in the image. Each image contains 780 features that were already processed into their numerical equivalent. Since the features are numbers between 0 and 255, the features were scaled by dividing by 255. Hence, the value of the features ranges from 0 to 1. The label comprises of 10 digits from 0 to 9. The dataset was stored in sparse format, which requires pre-processing before they could be deployed in ML algorithms. The preprocessing was done in R and the code used can be found in appendix A.

**DATA ANALYSIS**

**Activation Function**

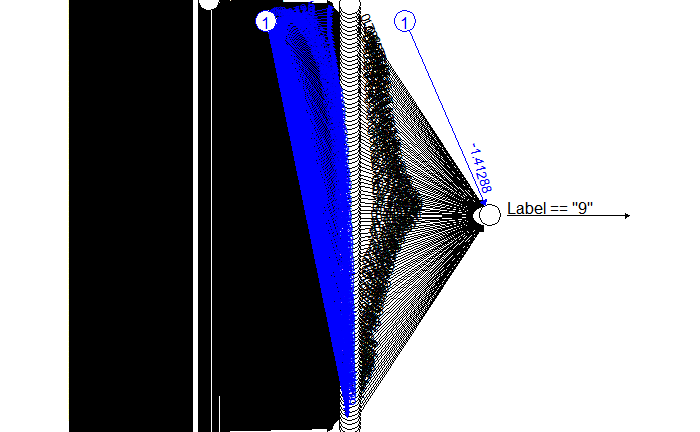
The RELU requires some extensive technical manipulation in R, which is outside the scope of this project. To overcome that challenge, a custom activation function called “softplus” was written to approximate the standard RELU. Figure 1 shows how softplus compares with the RELU. The code can be found in appendix B.



**Figure 1. Comparison between Softplus and RELU**

**Binary Classification (Neural Network)**

The processed MNIST dataset was split into training and testing data by ratio 70:30. Due to the ridiculous training time, the training data was sub-sampled. A two-layer NN for binary classification with a hidden layer comprising of 100 neurons and a singly output, as illustrated in Figure 2, was implemented. The binary classification was achieved by using the one-versus-the-rest rule. Simply put, a single label (i.e., 9) was arbitrarily selected from the 10 unique labels in the dataset. The Neural Network source code can be found in appendix B



**Figure 2. Binary Classification using Neural Network**

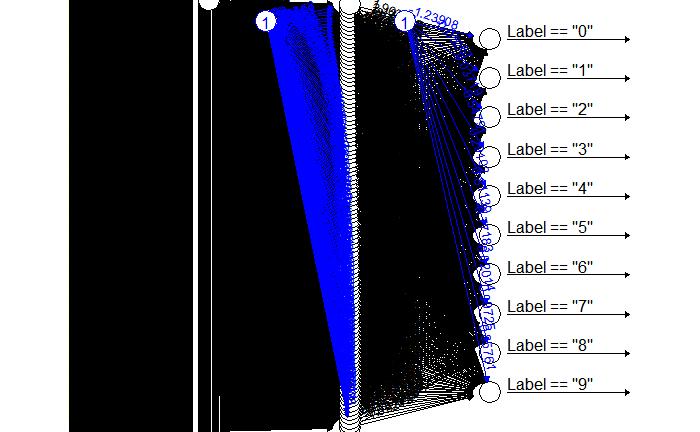
The model was evaluated by fitting the model on the test data. The accuracy of the model is 86.3% and the confusion matrix is as summarized in Table 1.

**Table 1. Confusion Matrix of the Binary Classifier**

|  |  |  |
| --- | --- | --- |
|  | FALSE | TRUE |
| FALSE | 15255 | 1005 |
| TRUE | 1455 | 285 |

**Multiclass Classification (Neural Network)**

The same training and test data were used to implement the multiclass classifier, as shown in Figure 3. The accuracy of the model is 92.83% and the confusion matrix is as summarized in Table 2.



**Figure 3. Multiclass Classification using Neural Network**

**Table 2. Confusion Matrix of the Multiclass Classifier**

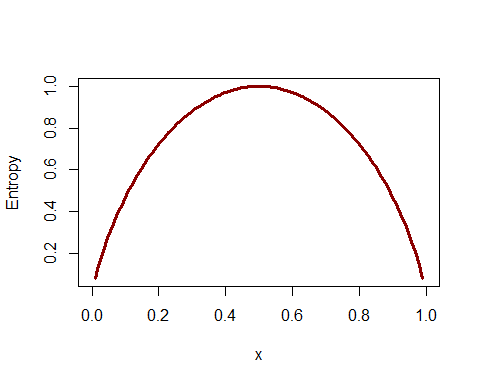
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (Actual/Predicted) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 1805 | 1 | 12 | 2 | 7 | 12 | 14 | 5 | 6 | 4 |
| 1 | 0 | 1946 | 25 | 9 | 8 | 11 | 3 | 9 | 17 | 5 |
| 2 | 9 | 7 | 1671 | 41 | 33 | 4 | 8 | 31 | 21 | 3 |
| 3 | 8 | 5 | 44 | 1600 | 6 | 40 | 6 | 29 | 46 | 15 |
| 4 | 6 | 8 | 8 | 2 | 1684 | 7 | 14 | 15 | 9 | 33 |
| 5 | 11 | 3 | 7 | 57 | 19 | 1430 | 29 | 6 | 30 | 11 |
| 6 | 17 | 6 | 14 | 0 | 8 | 30 | 1635 | 4 | 8 | 1 |
| 7 | 2 | 10 | 17 | 15 | 31 | 4 | 1 | 1729 | 6 | 36 |
| 8 | 9 | 23 | 18 | 25 | 18 | 42 | 14 | 11 | 1596 | 13 |
| 9 | 6 | 6 | 4 | 22 | 66 | 19 | 2 | 52 | 15 | 1548 |

**Direct Multiclass classification (Decision Tree)**

The decision tree algorithm implemented in the binary classification task is the C5.0 algorithm, which was developed by the famous computer scientist, J. Ross Quinlan as an improved version of C4.5 and the early Iterative Dichotomizer 3 (ID3).

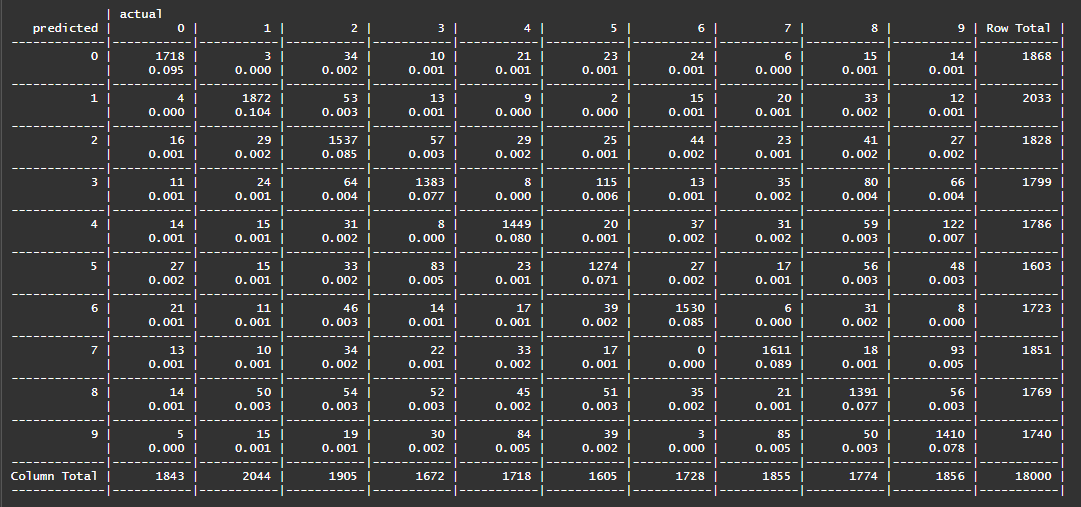
Choosing the best splitting candidate is a major challenge that determines the efficiency of a decision tree. C5.0 uses the entropy concept to split the trees. High entropy indicates a diverse subset that provides little information on which class the subset belongs to. Entropy can be mathematically expressed as shown in equation (1).

Where “S” represents the segment of data, “c” refers to the number of class levels, and “p­i” is proportion of values falling into the ith class level. A quick illustration of the concept of entropy is shown in Figure 4. To determine the optimal feature to split upon, the algorithm computes the difference in homogeneity from splitting on each feature, which is a measure of “information gain.” The higher the information gain, the better a feature is in producing homogenous subsets after splitting.



**Figure 4. Illustration of Entropy**

The confusion matrix of the decision tree implementation is shown in Figure 5. The resulting classification accuracy is approximately 84.3%.



**Figure 5. Confusion Matrix of the Multiclass Classification Decision Tree**

**CONCLUSION**

The analysis results showed that neural networks are powerful classifiers. However, comparing the classification accuracies, the vanilla decision tree algorithm is preferred because the interpretability is still within human comprehension, as compared to complete black-box models like neural networks.

**REFERENCES**

[1] Brett Lantz. (2015). Machine Learning with R. ISBN 978-1-78439-390-8

[2] <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist>

# APPENDIX: A

## Pre-Processing

read.libsvm = function( filename, dimensionality ) {  
   
 content = readLines(filename )  
 num\_lines = length( content )  
 yx = matrix( 0, num\_lines, dimensionality + 1 )  
   
 # loop over lines  
 for ( i in 1:num\_lines ) {  
   
 # split by spaces  
 line = as.vector( strsplit( content[i], ' ' )[[1]])  
   
 # save label  
 yx[i,1] = as.numeric( line[[1]] )  
   
 # loop over values  
 for ( j in 2:length( line )) {  
   
 # split by colon  
 index\_value = strsplit( line[j], ':' )[[1]]  
   
 index = as.numeric( index\_value[1] ) + 1 # +1 because label goes first

value = as.numeric( index\_value[2] )  
   
 yx[i, index] = value  
 }  
 }  
   
 return( yx )  
}

# APPENDIX: C

neuralnet <-

function (formula, data, hidden = 1, threshold = 0.01, stepmax = 1e+05,

rep = 1, startweights = NULL, learningrate.limit = NULL,

learningrate.factor = list(minus = 0.5, plus = 1.2), learningrate = NULL,

lifesign = "none", lifesign.step = 1000, algorithm = "rprop+",

err.fct = "sse", act.fct = "logistic", linear.output = TRUE,

exclude = NULL, constant.weights = NULL, likelihood = FALSE) {

# Save call

call <- match.call()

# Check arguments

if (is.null(data)) {

stop("Missing 'data' argument.", call. = FALSE)

}

data <- as.data.frame(data)

if (is.null(formula)) {

stop("Missing 'formula' argument.", call. = FALSE)

}

formula <- stats::as.formula(formula)

# Learning rate limit

if (!is.null(learningrate.limit)) {

if (length(learningrate.limit) != 2) {

stop("Argument 'learningrate.factor' must consist of two components.",

call. = FALSE)

}

learningrate.limit <- as.list(learningrate.limit)

names(learningrate.limit) <- c("min", "max")

if (is.na(learningrate.limit$min) || is.na(learningrate.limit$max)) {

stop("'learningrate.limit' must be a numeric vector",

call. = FALSE)

}

} else {

learningrate.limit <- list(min = 1e-10, max = 0.1)

}

# Learning rate factor

if (!is.null(learningrate.factor)) {

if (length(learningrate.factor) != 2) {

stop("Argument 'learningrate.factor' must consist of two components.",

call. = FALSE)

}

learningrate.factor <- as.list(learningrate.factor)

names(learningrate.factor) <- c("minus", "plus")

if (is.na(learningrate.factor$minus) || is.na(learningrate.factor$plus)) {

stop("'learningrate.factor' must be a numeric vector",

call. = FALSE)

}

} else {

learningrate.factor <- list(minus = 0.5, plus = 1.2)

}

# Learning rate (backprop)

if (algorithm == "backprop") {

if (is.null(learningrate) || !is.numeric(learningrate)) {

stop("Argument 'learningrate' must be a numeric value, if the backpropagation algorithm is used.",

call. = FALSE)

}

}

# TODO: Rename?

# Lifesign

if (!(lifesign %in% c("none", "minimal", "full"))) {

stop("Argument 'lifesign' must be one of 'none', 'minimal', 'full'.", call. = FALSE)

}

# Algorithm

if (!(algorithm %in% c("rprop+", "rprop-", "slr", "sag", "backprop"))) {

stop("Unknown algorithm.", call. = FALSE)

}

# Threshold

if (is.na(threshold)) {

stop("Argument 'threshold' must be a numeric value.", call. = FALSE)

}

# Hidden units

if (any(is.na(hidden))) {

stop("Argument 'hidden' must be an integer vector or a single integer.",

call. = FALSE)

}

if (length(hidden) > 1 && any(hidden == 0)) {

stop("Argument 'hidden' contains at least one 0.", call. = FALSE)

}

# Replications

if (is.na(rep)) {

stop("Argument 'rep' must be an integer", call. = FALSE)

}

# Max steps

if (is.na(stepmax)) {

stop("Argument 'stepmax' must be an integer", call. = FALSE)

}

# Activation function

if (!(is.function(act.fct) || act.fct %in% c("logistic", "tanh"))) {

stop("Unknown activation function.", call. = FALSE)

}

# Error function

if (!(is.function(err.fct) || err.fct %in% c("sse", "ce"))) {

stop("Unknown error function.", call. = FALSE)

}

# Formula interface

model.list <- list(response = attr(terms(as.formula(call("~", formula[[2]]))), "term.labels"),

variables = attr(terms(formula, data = data), "term.labels"))

response <- as.matrix(model.frame(as.formula(call("~", formula[[2]])), data))

covariate <- cbind(intercept = 1, as.matrix(data[, model.list$variables]))

# Multiclass response

if (is.character(response)) {

class.names <- unique(response[, 1])

response <- model.matrix( ~ response[,1]-1) == 1

colnames(response) <- class.names

model.list$response <- class.names

}

# Activation function

if (is.function(act.fct)) {

act.deriv.fct <- Deriv::Deriv(act.fct)

attr(act.fct, "type") <- "function"

} else {

converted.fct <- convert.activation.function(act.fct)

act.fct <- converted.fct$fct

act.deriv.fct <- converted.fct$deriv.fct

}

# Error function

if (is.function(err.fct)) {

attr(err.fct, "type") <- "function"

err.deriv.fct <- Deriv::Deriv(err.fct)

} else {

converted.fct <- convert.error.function(err.fct)

err.fct <- converted.fct$fct

err.deriv.fct <- converted.fct$deriv.fct

}

if (attr(err.fct, "type") == "ce" && !all(response %in% 0:1)) {

stop("Error function 'ce' only implemented for binary response.", call. = FALSE)

}

# Fit network for each replication

list.result <- lapply(1:rep, function(i) {

# Show progress

if (lifesign != "none") {

lifesign <- display(hidden, threshold, rep, i, lifesign)

}

# Fit network

calculate.neuralnet(learningrate.limit = learningrate.limit,

learningrate.factor = learningrate.factor, covariate = covariate,

response = response, data = data, model.list = model.list,

threshold = threshold, lifesign.step = lifesign.step,

stepmax = stepmax, hidden = hidden, lifesign = lifesign,

startweights = startweights, algorithm = algorithm,

err.fct = err.fct, err.deriv.fct = err.deriv.fct,

act.fct = act.fct, act.deriv.fct = act.deriv.fct,

rep = i, linear.output = linear.output, exclude = exclude,

constant.weights = constant.weights, likelihood = likelihood,

learningrate.bp = learningrate)

})

matrix <- sapply(list.result, function(x) {x$output.vector})

if (all(sapply(matrix, is.null))) {

list.result <- NULL

matrix <- NULL

ncol.matrix <- 0

} else {

ncol.matrix <- ncol(matrix)

}

# Warning if some replications did not converge

if (ncol.matrix < rep) {

warning(sprintf("Algorithm did not converge in %s of %s repetition(s) within the stepmax.",

(rep - ncol.matrix), rep), call. = FALSE)

}

# Return output

generate.output(covariate, call, rep, threshold, matrix,

startweights, model.list, response, err.fct, act.fct,

data, list.result, linear.output, exclude)

}

# Display output of replication

display <- function (hidden, threshold, rep, i.rep, lifesign) {

message("hidden: ", paste(hidden, collapse = ", "), " thresh: ",

threshold, " rep: ", strrep(" ", nchar(rep) - nchar(i.rep)),

i.rep, "/", rep, " steps: ", appendLF = FALSE)

utils::flush.console()

if (lifesign == "full") {

lifesign <- sum(nchar(hidden)) + 2 \* length(hidden) -

2 + max(nchar(threshold)) + 2 \* nchar(rep) + 41

}

return(lifesign)

}

# Generate output object

generate.output <- function(covariate, call, rep, threshold, matrix, startweights,

model.list, response, err.fct, act.fct, data, list.result,

linear.output, exclude) {

nn <- list(call = call, response = response, covariate = covariate[, -1, drop = FALSE],

model.list = model.list, err.fct = err.fct, act.fct = act.fct,

linear.output = linear.output, data = data, exclude = exclude)

if (!is.null(matrix)) {

nn$net.result <- lapply(list.result, function(x) {x$net.result})

nn$weights <- lapply(list.result, function(x) {x$weights})

nn$generalized.weights <- lapply(list.result, function(x) {x$generalized.weights})

nn$startweights <- lapply(list.result, function(x) {x$startweights})

nn$result.matrix <- matrix

rownames(nn$result.matrix) <- c(rownames(matrix)[rownames(matrix) != ""],

get\_weight\_names(nn$weights[[1]], model.list))

}

class(nn) <- c("nn")

return(nn)

}

# Get names of all weights in network

get\_weight\_names <- function(weights, model.list) {

# All hidden unit names

if (length(weights) > 1) {

hidden\_units <- lapply(1:(length(weights) - 1), function(i) {

paste0(i, "layhid", 1:ncol(weights[[i]]))

})

} else {

hidden\_units <- list()

}

# All unit names including input and output

units <- c(list(model.list$variables),

hidden\_units,

list(model.list$response))

# Combine each layer with the next, add intercept

weight\_names <- do.call(c, lapply(1:(length(units) - 1), function(i) {

as.vector(outer(c("Intercept", units[[i]]), units[[i + 1]], paste, sep = ".to."))

}))

return(weight\_names)