Model 2

SITTY AZQUIA M. CAMAMA

2022-12-15

Network-Based Classification Model

In this document, we will perform Network-Based Classification Model using **radiomics data**.

Load Helper and Model Packages

```
library(dplyr)         # for data manipulation
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(keras)         # for fitting DNNs
library(tfruns)        # for additional grid search & model training functions
library(tensorflow)
library(tfestimators)  # provides grid search & model training interface
```

```
## tfestimators is not recomended for new code. It is only compatible with Tensorflow version 1, and is
```

```
library(rsample)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
```

```
## v ggplot2 3.4.0     v purrr   0.3.4
## v tibble  3.1.8     v stringr 1.4.1
## v tidyr   1.2.1     v forcats 0.5.2
## v readr   2.1.3
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(bestNormalize)
```

Load Data Sets

Radiomics data contains 197 rows and 431 columns: **Failure.binary**: binary property to predict

```
radiomicsdata <- read.csv("~/R CLASS/FINAL PROJECT/radiomics_completedata.csv")
View(radiomicsdata)
```

Data Pre-Processing

Check for null and missing values

Using **anyNA()** function, We can determine if any missing values in our data. The result shows either **TRUE** or **FALSE**. If true, omit the missing values using **na.omit().** Hence, our data has no missing values.

```
anyNA(radiomicsdata)
```

```
## [1] FALSE
```

Check for normality

The **Shapiro-Wilk's Test** is used to check the normality of the data. The null hypothesis states that data are normally distributed. Before, we test the normality, remove the categorical and binary variable.

```
rd <- radiomicsdata%>%select_if(is.numeric)
rd <- rd[,-1]
test <- apply(rd,2,function(x){shapiro.test(x)})
```

unlist() function is used to convert a list to vector, so we can have the list of p-value of all variables.

```
pvalue_list <- unlist(lapply(test, function(x) x$p.value))
```

Compute the sum of total variable with p-value less than 0.05 alpha. Thus, we have 428 variables that are not normally distributed and Entropy_cooc.W.ADC is normally distributed.

```
sum(pvalue_list<0.05)  # not normally distributed
```

```
## [1] 428
```

```
sum(pvalue_list>0.05)  # normally distributed
```

```
## [1] 1
```

```
test$Entropy_cooc.W.ADC
```

```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.98903, p-value = 0.135
```

To normalized the data, remove first the categorical, binary and Entropy_cooc.W.ADC variable and use **orderNorm()** function. The **x.t** is the elements of orderNorm() function transformed original data.

```
rdnorm=radiomicsdata[,c(3,5:length(names(radiomicsdata)))]
rdnorm=apply(rdnorm,2,orderNorm)
rdnorm=lapply(rdnorm, function(x) x$x.t)
rdnorm=rdnorm%>%as.data.frame()
```

Test again using shapiro-wilk's test.

```
test2=apply(rdnorm,2,shapiro.test)
pvalue_list2=unlist(lapply(test2, function(x) x$p.value))
```

Compute the sum of total variable with p-value less than 0.05 alpha and more than 0.05 alpha. Finally, our data is normally distributed.

```
sum(pvalue_list2<0.05)    # not normally distributed
```

```
## [1] 0
```

```
sum(pvalue_list2>0.05)    # normally distributed
```

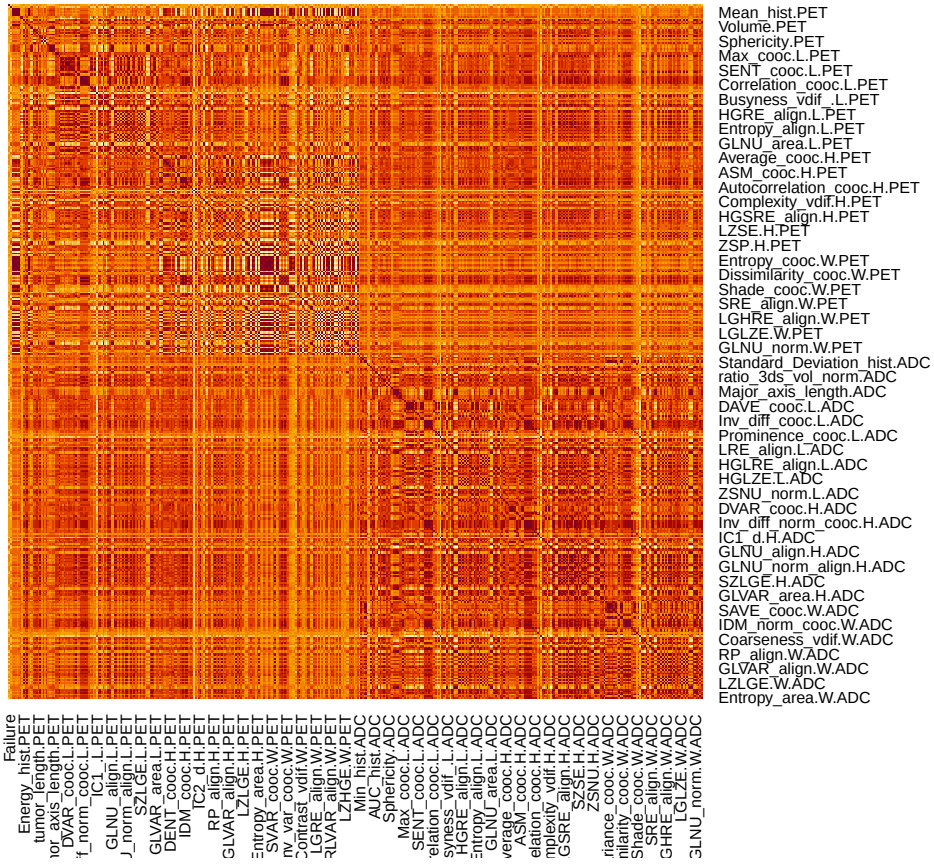
```
## [1] 428
```

Create new data with the **Failure.binary**, **Entropy_cooc.W.ADC**, and **rd2** variables.

```
keep = select(radiomicsdata, c("Failure.binary",  "Entropy_cooc.W.ADC"))
newdata = cbind(keep,rdnorm)
```

Get the correlation of the whole data expect the categorical variables

```
CorMatrix=cor(newdata[,-c(1,2)])
heatmap(CorMatrix,Rowv=NA,Colv=NA,scale="none",revC = T)
```



Splitting

Split the data into training (80%) and testing (30%).

```
newdata<-newdata %>%
  mutate(Failure.binary=ifelse(Failure.binary== "No",0,1))

set.seed(123)
split = initial_split(newdata,prop = 0.8 ,strata = "Failure.binary")
split_train <- training(split)
```

```
split_test  <- testing(split)

xtrain <- split_train[,-c(1,2)]%>%as.matrix.data.frame()
xtest <- split_test[,-c(1,2)]%>%as.matrix.data.frame()
ytrain <- split_train$Failure.binary
ytest <- split_test$Failure.binary
```

Reshaping the dataset

```
xtrain <- array_reshape(xtrain, c(nrow(xtrain), ncol(xtrain)))
xtrain <- xtrain

xtest <- array_reshape(xtest, c(nrow(xtest), ncol(xtest)))
xtest <- xtest

ytrain <- to_categorical(ytrain, num_classes = 2)
```

```
## Loaded Tensorflow version 2.9.3
```

```
ytest <- to_categorical(ytest, num_classes = 2)
```

Run the model

with the R function **keras_model_sequential()** of keras package, allows us to create our network with a layering approach. So, we are going to create five hidden layers with 256, 128, 128, 64 and 64 neurons, respectively with activation functions **Sigmoid** and 2 neurons for output layer with activation functions of **Softmax**. Every layer is followed by a dropout to avoid overfitting.

```
model <- keras_model_sequential() %>%

   # Network architecture
  layer_dense(units = 256, activation = "sigmoid", input_shape = c(ncol(xtrain))) %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "sigmoid") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = "softmax")
```

Backpropagation Compiler Approach

```
model %>%

compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy")
  )
```

Trained the model

We've already built a fundamental model; all that remains is to feed it some data to train on. To achieve this, we input our training data and model into a **fit()** function.

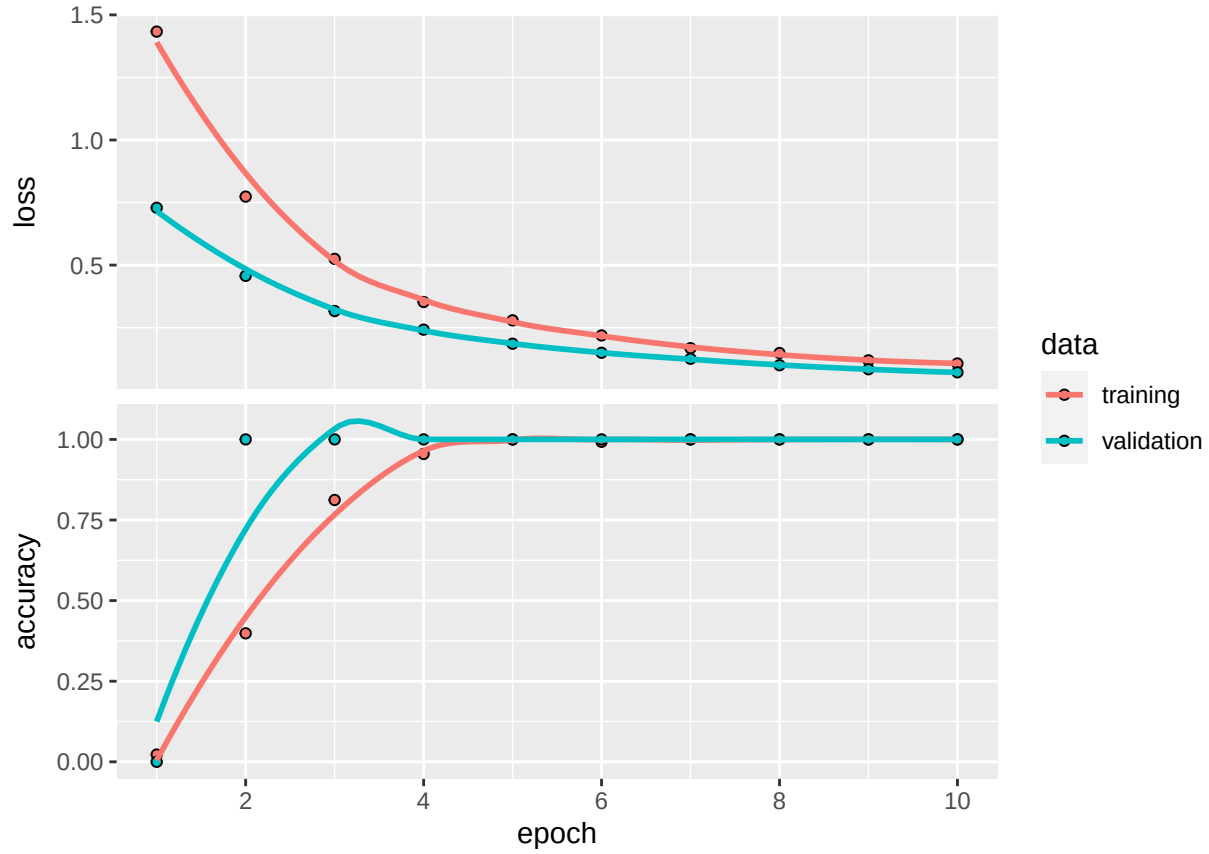
An epoch indicates how many times the algorithm views the entire dataset. Therefore, an epoch has ended whenever the algorithm has viewed all of the samples in the data set. Since a single epoch would be too large to transmit to the computer all at once, we divide it in several smaller batches.

```
fitrd <- model %>%
  fit(xtrain, ytrain,
      epochs = 10,
      batch_size = 128,
      validation_split = 0.15)

# Display output
fitrd
```

```
##
## Final epoch (plot to see history):
##         loss: 0.1069
##     accuracy: 1
##     val_loss: 0.07147
## val_accuracy: 1
```

```
#plot the training and validation performance over 10 epochs
plot(fitrd)
```



Evaluate the trained model using testing dataset

```
model %>%
  evaluate(xtest, ytest)
```

```
##       loss   accuracy
## 0.07003044 1.00000000
```

```
dim(xtest)
```

```
## [1]  40 428
```

```
dim(ytest)
```

```
## [1] 40  2
```

Model prediction using testing dataset

```
model   %>% predict(xtest) %>% `>`(0.5) %>% k_cast("int32")
```

```
## tf.Tensor(
## [[0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
```

```
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]
##  [0 1]], shape=(40, 2), dtype=int32)
```