# ***Module 1:***

## **Basics**

ls() to see the list in the Environment Pane

objects() similar to ls() [variable, dataset, paragraph of text, or mathematical index]

vec %\*% vec matrix operations need to be specified in this particular manner

matrix(#s, nrow, ncol, byrow = T/F) generate a matrix

data.frame(var =

sample(x,

size = #,

replace = T/F) generate a data frame with randomly sampling/selecting

mean(df$var) reading the mean of the variable

[median, range, quantile(var, quantile), IQR, var, sd, summary]

runif(#, min, max) randomly selecting numbers between the min and max # of times

## **Descriptive Measures**

table(df$var) calculates the mode or count for each category in the var

prop.table(table(df$var) calculating the proportions

xtabs(~var, data = df) calculating the mode or count for each category in the var

tapply(X = var1, INDEX = var2,

FUN = “mean”) mean of X for each index

**- library(skimr)**

skim(df)

## **Explore**

head(df, n = #) reads the first couple of rows in the data frame

tail(df, n = #) reads the last couple of rows in the data frame

str(df) details of the data frame

nrow(df) number of rows in the data frame

ncol(df) number of columns in the data frame

dim(df) number of rows and columns in the data frame

names(df) variable names in the data frame

df$var>logical logical output

df[df$var>logical, c(var1,,)] outputs results of var1 when logical is true

order(df$var, decreasing =F) sorting the results of var in ascending order

rownames(data)

**- library(dplyr); library(tidyr)**

select(starts\_with()) selecting variables that need to be used for the data frame

filter() filtering out results from some variables

mutate() modifies the current variable or creates a new variable (or column)

arrange(desc()) sorting the variable in descending order if including desc()

bind\_cols() combine columns from several data frames

group\_by() groups by the variables

ungroup() removes grouping

summarize() n() counts the number of occurrences in the dataset

gather() wide to tall data frame

rename(NEW = var) rename column name

count() counts the occurrences in each category of the variable

pivot\_longer(cols,

names\_to,

values\_to) expanding the variable names into categories of the variable

spread() tall to wide data frame

pivot\_wider(

names\_from,

values\_from) shrinking the categories of the variables

summarise\_all(.funs) summarizing for all the variables based on the function

## **Visual Summary**

**- library(ggplot2)**

ggplot(data = df, mapping = aes(x = var1, y = var2, fill/color = factor(var)) +

geom\_histogram(binwidth = #) plotting histogram

hist(df$var) plotting histogram

geom\_freqpoly(size = #) plotting frequency polygon

geom\_density(size = #) plotting density curve

geom\_boxplot(outlier.color, fill, color) plotting boxplot

geom\_text(data = df[var>logical,], aes(label = var1), nudge\_x = #)

adding text to the graphs

geom\_qq\_line() construct a normality line

geom\_qq() plot the qqnorm

geom\_bar(stat = “summary”, fun = “”,

show.legend = T/F,

position = c(

“dodge”,

“stack”,

“fill”,

“identity”) plot a bar chart

coord\_flip() flip the axes

geom\_point() plot the points from the data frame

geom\_smooth(method = “lm”,

se = T/F, size = #) plot the line

facet\_grid(var1~var2 / var2 ~ var1) plot multiple combinations

facet\_wrap(~var1, scales = “free”) plot multiple combinations

# ***Module 2 & 3:***

## **Data Parsing**

**- library(readr)**

parse\_number(“”, locale = locale(grouping\_mark = “.”)) extracts only the number

parse\_character(var, trim\_ws = T) parses unusual strings

**- library(stringr)**

str\_trim(var, side = c(“”)) remove white spaces on sides

**- library(lubridate)**

Dates Parsing – read the section for proper formatting

as.Date(“Month Day, Year”, format = “%b %d, %Y”) reading the string as date with format

## **Joining Tables**

**- library(dplyr)**

inner\_join(data, by = c()) joins 2 datasets by few variable

## **Missing Data**

for(lv in #:#){

} for loop to fill in the missing values

When the dataset has a reasonable number of observations and variables, a simple method to visualize missing information is a heatmap (check the red/orange and teal map of bricks)

Another way to spot patterns in missing data is creating tiles to the heatmap

**- library(ComplexHeatmap); library(circlize)**

sapply(data,

function(x) sum(is.na(x))) number of missing values for each variable

missing[!sapply(data, function(x) any(is.na(x))), ] deleting rows with missing values

encoded[is.na(data)] 🡨 NA encode the missing values

**- library(mice)**

mice::complete(mice(missing, method = “rf”, seed)) predicting values for NA

**- library(caret)**

predict(preProcess(missing, method = “bagImpute”), newdata) predicting values for NA

## **Variable Transformations**

**- library(forcats)**

fct\_count(df$var) counts the occurrences of var factors

fct\_collapse(df$var, “new” = c(“subbing factors”) removes some factors of the variable

fct\_recode(.f = df$var, “old” = “NEW”) rename the factors in the variable

**- library(e1071)**

preProcess(df[,c(“”, )], method = “BoxCox”) box-cox transform skewness to normal dist.

preProcess(df[,c(“”, )], method = “BoxCox”)$bc results with estimating lambda

apply(X=df[,c(“”,)], MARGIN = 2, FUN)

**library(Hmisc)**

# ***Module 4:***

## **Linear Regression**

cor(data$var1, data$var2) correlation coefficient

lm(y ~ x, data) linear model

paste(“y”, “=”, round(coef(model1)[1], 0), “+”, round(coef(model1)[2], 0), “x”)

summary(lm(y~x)) summary of the model

predict(model1) predicting values using x and the model

sse = sum((pred – train$y)^2) sum of squared error use to calculate R^2

sst = sum((mean(train$y) – train$y)^2) sum of squared totals to calculate R^2

model\_r2 = 1 – sse/sst; model\_r2 R^2 value of the model

rmse = sqrt(mean((pred – train$y)^2)

rmse = RMSE(pred, obs)

rmse = sqrt(sum((pred – obs)^2 / nrow(data))) Root mean square error of the model

predict(model1, newdat = data.frame(x = #) the model’s y value for a given x value

anova(model1) anova model including residuals

lm(y ~ poly(x, #), data) polynomial regression where # = deg of poly

predict(model, newdata = test) predicting values for test sample

- **library(lm.beta)**

lm.beta(model) standardized coefficients on the impact

## **Modelling Frameworks**

- **Simple Random Sampling: library(caret)**

split = sample(x = 1:nrow(data), size = 0.8\*nrow(data))

train = data[split, ]

test = data[-split, ]

- **Stratified Sampling: library(caret) with numeric outcome variable**

split = createDataPartition(y = data$resp\_var, p = 0.8, list = F, groups = #)

train = data[split, ]

test = data[-split, ]

- **Stratified Sampling: library(caret) with categorical outcome variable**

data$cat\_var = ifelse(…, cat1, cat2)

split = createDataPartition(y = data$cat\_var, p = 0.8, list = F, groups = #)

train = data[split, ]

test = data[-split, ]

table(test$cat\_var)

prop.table(rbind(train = table(train$cat\_var),

test = table(test$cat\_var)),

margin = 1) compare proportions in each sample

- **Stratified Sampling: library(caTools) with categorical outcome**

split = sample.split(Y = data$resp\_var, SplitRatio = 0.8)

train = data[split, ]

test = data[!split, ]

table(test$y)

prop.table(rbind(train = table(train$y),

test = table(test$y)),

margin = 1) compare proportions in each sample

# ***Module 5:***

## **Logistic Regression**

p(X) = e^(b0 + b1X) / (1 + e^(b0 + b1X)

p(X) / (1 – p(X)) = e^(b0 + b1X)

ln(p(X) / (1 – p(X)) = b0 + b1X

glm(y ~ x, data, family = “binomial”) if the outcome is binary

exp(model$coef[1] + model1$coef[2]\*200)/(1+exp(model1$coef[1] + model1$coef[2]\*200))

predict(model, newdata = data.frame(x = #), type = “response”)

summary(model)$aic lower AIC, better the model

summary(model)$deviance

-2\*logLik(model) measure of error in the model

- **Difference of error between baseline (null model) and new model**

model\_chi\_square = model$null.deviance - model$deviance

df = model$df.null - model$df.residual

p\_val = pchisq(model$null.deviance - model$deviance,

df=model$df.null - model$df.residual,

lower.tail=FALSE)

paste('Chi-square value of',model\_chi\_square , 'with',df , 'df', 'corresponding to a p-value of', p\_val )

- **Mimic R2 from linear regression: McFadden R2, Cox and Snell R2, and Nagleklerke R2**

**library(descr)**

dev = model$deviance

nulldev = model$null.deviance

n = nrow(train)

mcfadden\_r2 = 1 - dev/nulldev

cox\_and\_snell\_r2 = 1 - exp(-(nulldev - dev)/n)

nagelklerke\_r2 = cox\_and\_snell\_r2/(1 - exp(-nulldev/n))

mcfadden\_r2; cox\_and\_snell\_r2; nagelklerke\_r2

LogRegR2(model) lists Chi2, df, sig, and above 3 R2’s

ct = table(name1 = y,

name2(pred) = as.numeric(pred > 0.5)) classification table or confusion matrix

accuracy = sum(ct[1,1],ct[2,2])/nrow(train) accuracy of the predictions

specificity = ct[1,1]/sum(ct[1,1],ct[1,2]) specificity

sensitivity = ct[2,2]/sum(ct[2,1],ct[2,2]) sensitivity

- **library(ROCR)**

ROCRpred = prediction(pred, y)

ROCRperf = performance(ROCRpred, “tpr”, “fpr”)

Plot(ROCRperf)

Plot(ROCRperf, colorize = T, print.cutoffs.at = seq(0,1,0.2), text.adj = c(-0.3,2),

xlab = “1 – Specificity”, ylab = “Sensitivity”)

auc = as.numeric(performance(ROCRpred, “auc”)@y.values) auc measure

ROC for a baseline model will look like

baselinePred = pred\*0

ROCRpred = prediction(baselinePred,test$sold)

ROCRperf = performance(ROCRpred,"tpr","fpr")

plot(ROCRperf,xlab="1 - Specificity",ylab="Sensitivity") # relabeled axes

# ***Module 6:***

## **Feature Selection**

- **Correlation Matrix; library(tidyr); library(dplyr); library(ggplot2)**

cor(train[,-y])

corMatrix = as.data.frame(cor(train[,-y]))

corMatrix$var1 = rownames(corMatrix)

corMatrix %>%

gather(key=var2,value=r,1:length in var1)%>%

arrange(var1,desc(var2))%>%

ggplot(aes(x=var1,y=reorder(var2, order(var2,decreasing=F)),fill=r))+

geom\_tile()+

geom\_text(aes(label=round(r,2)),size=3)+

scale\_fill\_gradientn(colours = c('#d7191c','#fdae61','#ffffbf','#a6d96a','#1a9641'))+

theme(axis.text.x=element\_text(angle=75,hjust = 1))+xlab('')+ylab('')

**- Correlation Heat Map; library(ggcorrplot)**

ggcorrplot(cor(train), method = 'square', type = 'lower', show.diag = F,

colors = c('#e9a3c9', '#f7f7f7', '#a1d76a'))

**- library(car)**

vif(model)

**- Best Subset Selection; library(leaps)**

subsets = regsubsets(y~., data = train, nvmax = up to # of variables creating models)

summary(subsets)

subsets\_measures = data.frame(model = 1:length(summary(subsets)$cp),

cp = summary(subsets)$cp,

bic = summary(subsets)$bic,

adjr2 = summary(subsets)$adjr2)

coef(subsets, which.min(summary(subsets)$cp)) #which.min(summary(subsets)$cp)

**- Forward Selection**

start\_mod = lm(y~1, data = train)

empty\_mod = lm(y~1, data = train)

full\_mod = lm(y~., data = train)

forwardStepwise = step(start\_mod,

scope = list(upper = full\_mod, lower = empty\_mod),

direction = 'forward')

**- Backward Selection**

start\_mod = lm(y~., data = train)

empty\_mod = lm(y~1, data = train)

full\_mod = lm(y~., data = train)

backwardStepwise = step(start\_mod,

scope = list(upper = full\_mod, lower = empty\_mod),

direction = 'backward')

**- Hybrid Stepwise Variable Selection**

start\_mod = lm(y~1, data = train)

empty\_mod = lm(quality~1,data = train)

full\_mod = lm(quality~., data = train)

hybridStepwise = step(start\_mod,

scope = list(upper = full\_mod, lower = empty\_mod),

direction = 'both')

## **Shrinkage Methods**

**- Ridge Regression: shrinking the coefficients as close to 0 as possible**

library(glmnet)

x = model.matrix(y~.-1, data = train)

y = train$y

set.seed(617)

ridge = glmnet(x = x, y = y, alpha = 0)

**- Lasso Regression: shrinking the coefficients to 0**

cv\_lasso = cv.glmnet(x = x, y = y, alpha = 1, type.measure = 'mse')

cv\_lasso$lambda.min

coef(cv\_lasso, s = cv\_lasso$lambsa.1se) %>% round(4)

## **Dimension Reduction**

trainPredictors = train[, -y]

testPredictors = test[, -y]

pca = prcomp(trainPredictors, scale. = T)

train\_components = data.frame(cbind(pca$x[,1:6], quality = train$y))

train\_model = lm(y~., train\_components)

summary(train\_model)

test\_pca = predict(pca, newdata = testPredictors)

test\_components = data.frame(cbind(test\_pca[,1:6], quality = test$y))

pred = predict(train\_model, newdata = test\_components)

sse = sum((pred-test\_components$quality)^2)

sst = sum((mean(train\_components$quality) - test\_components$quality)^2)

r2\_test = 1 - sse/sst

r2\_test

# ***Module 8:***

## **Regression Trees – outcome is continuous**

- **library(rpart); library(rpart.plot)**

MSE should decrease

 minsplit: the minimum number of observations that must exist in a node in order for a split to be attempted; default = 20

 minbucket: the minimum number of observations in any terminal node; default = 6 (20/3)

 maxdepth: the maximum depth of any node of the final tree

 cp: complexity parameter; default = 0.01

tree = rpart(y ~ x, data = train, method = 'anova',

control = rpart.control(c(minsplit = #,

minbucket = #,

maxdepth = #,

cp = #) anova variation in branches

summary(tree)

prp(tree)

rpart.plot(tree)

predict(tree, newdata = data.frame(x = #), type = “vector”)

tree$variable.importance

**- library(Metrics)**

rmse function same as RMSE function except rmse(actual, predicted)

## **Classification Trees – outcome is categorical**

- **library(rpart); library(rpart.plot)**

 minsplit: the minimum number of observations that must exist in a node in order for a split to be attempted; default = 20

 minbucket: the minimum number of observations in any terminal node; default = 7 (20/3)

 maxdepth: the maximum depth of any node of the final tree

 cp: complexity parameter; default = 0.01

tree = rpart(y~x, data = train, method = 'class',

control = rpart.control(c(minsplit = #,

minbucket = #,

maxdepth = #,

cp = #))

summary(tree)

prp(tree)

rpart.plot(tree)

predict(tree, newdata = data.frame(x = #), type = “prob”) predicted prob for each class

predict(tree, newdata = data.frame(x = #), type = “class”) predicted class

predict(tree, newdata = data.frame(x = #), type = “prob”)[, “high”] > 0.5

unique(predict(tree, type = “prob”))

tree$variable.importance

pred = predict(tree, type = “prob”)[, 2]

pred\_class = factor(ifelse(pred2 > 0.5, 'high', 'low'), levels = c('low', 'high'))

ct = table(train$y, pred\_class)

**- library(caret)**

confusionMatrix(data = pred\_class, reference = train$y, positive = “high”)

# ***Module 9:***

## **Advanced Trees**

**- Default Tree**

tree\_baseline = rpart(y ~., data = train, method = “anova”)

pred\_train = predict(tree\_baseline)

- **Maximal Tree: largest tree possible**

maximalTree = rpart(earn ~., data = train, control = rpart.control(cp = 0))

**- Tuned Tree: tune hyperparameters to address the threat of overfitting; library(caret)**

Example: tune complexity of tree using 5-fold cross-validation. Examine cv error for 100 cp values

trControl = trainControl(method = 'cv', number = 5)

tuneGrid = expand.grid(.cp = seq(from = 0.001, to = 0.1, by = 0.001))

set.seed(617)

cvModel = train(y~.,

data=train,

method="rpart",

trControl = trControl,

tuneGrid = tuneGrid)

cvModel$results

Evaluate Tuned model on Test sample

cvTree = rpart(y~., data = train, cp = cvModel$bestTune$cp)

plot(cvTree)

cvTree$bestTune

**- Bag Models: generate a large number of bootstrapped samples. Tree is fit to each sample; library(randomForest)**

bag = randomForest(y ~., data = train, mtry = ncol(train) - 1, ntree = 1000)

hist(treesize(bag)); plot(bag)

importance(bag)

**- Random Forest Model: considers a subset of predictors unlike bag model; library(randomForest)**

forest = randomForest(y~., data = train, ntree = 1000)

plot(forest)

varImpPlot(forest)

**- Tuned Random Forest: mtry parameter can be tuned to improve model predictions**

trControl=trainControl(method = "cv", number=5)

tuneGrid = expand.grid(mtry = 1:ncol(train) - 1)

set.seed(617)

cvModel = train(y~., data = train,

method = "rf", ntree = 1000, trControl = trControl, tuneGrid = tuneGrid )

cvModel

**- Forest with Ranger: popular for running random forest model; library(ranger)**

forest\_ranger = ranger(y~., data = train, num.trees = 1000)

pred = predict(forest\_ranger, data = test, num.trees = 1000)

**- Tuned Forest Ranger: for different values of mtry, splitrule, min.node.size**

trControl=trainControl(method="cv",number=5)

tuneGrid = expand.grid(mtry=1:4,

splitrule = c('variance','extratrees','maxstat'),

min.node.size = c(2,5,10,15,20,25))

set.seed(617)

cvModel = train(earn~.,

data=train,

method="ranger",

num.trees=1000,

trControl=trControl,

tuneGrid=tuneGrid )

cv\_forest\_ranger = ranger(earn~.,

data=train,

num.trees = 1000,

mtry=cvModel$bestTune$mtry,

min.node.size = cvModel$bestTune$min.node.size,

splitrule = cvModel$bestTune$splitrule)

pred = predict(cv\_forest\_ranger, data =test, num.trees = 1000)

**- Boosting Models: derive predictions from a number of trees, difference is that trees are grown sequentially, each tree is grown using info from previous grown trees, can be overfitting the data; library(gbm)**

set.seed(617)

boost = gbm(earn~.,

data=train,

distribution="gaussian",

n.trees = 500,

interaction.depth = 2,

shrinkage = 0.01)

pred = predict(boost, n.trees = 500)

pred = predict(boost, newdata=test, n.trees = 500)

- **Tuned gbm; library(caret)**

set.seed(1031)

trControl = trainControl(method="cv",number=5)

tuneGrid = expand.grid(n.trees = 500,

interaction.depth = c(1,2,3),

shrinkage = (1:100)\*0.001,

n.minobsinnode=c(5,10,15))

garbage = capture.output(cvModel <- train(Balance~.,

data=train,

method="gbm",

trControl=trControl,

tuneGrid=tuneGrid))

set.seed(1031)

cvboost = gbm(Balance~.,

data=train,

distribution="gaussian",

n.trees=500,

interaction.depth=cvModel$bestTune$interaction.depth,

shrinkage=cvModel$bestTune$shrinkage,

n.minobsinnode = cvModel$bestTune$n.minobsinnode)

pred\_train = predict(cvboost, n.trees=500)

**- Boosting with cv: tuning the above overfitted model; library(caret)**

set.seed(617)

trControl = trainControl(method="cv",number=5)

tuneGrid = expand.grid(n.trees = 500,

interaction.depth = c(1,2,3),

shrinkage = (1:100)\*0.001,

n.minobsinnode=c(5,10,15))

garbage = capture.output(cvModel <- train(earn~.,

data=train,

method="gbm",

trControl=trControl,

tuneGrid=tuneGrid))

set.seed(617)

cvBoost = gbm(earn~.,

data=train,

distribution="gaussian",

n.trees=cvModel$bestTune$n.trees,

interaction.depth=cvModel$bestTune$interaction.depth,

shrinkage=cvModel$bestTune$shrinkage,

n.minobsinnode = cvModel$bestTune$n.minobsinnode)

pred = predict(cvBoost,test,n.trees=500)

- **Boosting with xgboost: optimized distributed gradient boosting library; highly efficient, flexible, portable; library(vtreat)**

trt = designTreatmentsZ(dframe = train,

varlist = names(train)[2:6])

newvars = trt$scoreFrame[trt$scoreFrame$code%in% c('clean','lev'),'varName']

train\_input = prepare(treatmentplan = trt, dframe = train,

varRestriction = newvars)

test\_input = prepare(treatmentplan = trt, dframe = test,

varRestriction = newvars)

**xgboost can overfit the train data, so we will identify the optimal nrounds, use cv within model; library(xgboost); library(caret)**

tune\_nrounds = xgb.cv(data=as.matrix(train\_input),

label = train$earn,

nrounds=250,

nfold = 5,

verbose = 0)

which.min(tune\_nrounds$evaluation\_log$test\_rmse\_mean)

xgboost2= xgboost(data=as.matrix(train\_input),

label = train$earn,

nrounds=6,

verbose = 0)

pred = predict(xgboost2,

newdata=as.matrix(test\_input))