

Feature Selection and Predictive Performance in Incomplete, Higher-Dimensional Datasets

Method 3: RF - Random Forest with NA Roughfix Imputation

Scenario 38: General Overview

The second alternative method is Random Forest ("RF") using the `na.roughfix` option for missing data imputation. The basis for this alternative method comes from the observation in Liu et. al. (2016) regarding similarities between MIRL and RF. Instead of using a multiple imputation method, Random Forest uses feature medians to impute missing values.

Initial Setup

Load relevant packages, set seed and other scenario parameters.

```
packs          = c("doParallel", "dplyr", "formattable", "kableExtra", "mltools", "randomForest", "tidyr")
lapply(packs, require, character.only = TRUE)
seedNum        = 103871
set.seed(seedNum)
scenID         = 38
scenName       = "RF"
```

Turn on Parallelization

```
cl              = makeCluster(20)
registerDoParallel(cl)
```

Load Train and Test Data

Load simulated datasets created in "1-SimulateDatasets.RMD".

```
load(paste0("trainDataSet", scenID, ".Rdata"))
load(paste0("testDataSet", scenID, ".Rdata"))
p      = dim(trainDataSet[[1]])[2]-1
# separate supervisor from explanatory variables
yTrain = lapply(trainDataSet, function(x) x[, (p+1)])
yTest  = lapply(testDataSet,  function(x) x[, (p+1)])
```

Random Forest

Rather than applying regression based methods, this approach utilizes tree based methods. An initial random forest is fit using all the features to perform feature selection. Unimportant features are then removed and a random forest is refit using only the selected features, which is used for the supervisor prediction.

```
outrfDataSet    = vector("list", 100)
yHatRF          = matrix(0, nrow=100, ncol=200)
mseRF           = rep(0,100)
varImpTypeOne   = varImpTypeTwo = matrix(0, nrow=100, ncol=p)

for (i in 1:100){
# Random Forest
  outrfDataSet[[i]] = randomForest(y~., data = trainDataSet[[i]], ntree = 400, importance = TRUE, mtry = p/3, na.action = na.roughfix)
# variable importance
  varImpTypeOne[i,] = importance(outrfDataSet[[i]], type = 1)
# prediction
  yHatRF[i,] = predict(outrfDataSet[[i]], newdata = na.roughfix(testDataSet[[i]]), type = "response")
  mseRF[i] = mean((yHatRF[,i]-yTest[[i]])^2)
}
```

Feature importance is used for feature selection and an initial indication of predictive performance is obtained below:

```
cat(paste("RMSE for RF using all predictors:"),round(sqrt(mean(mseRF)),2))
```

```
## RMSE for RF using all predictors: 1.53
```

```
topTenTypeOne = apply(as.data.frame(varImpTypeOne),1, function(x)names(sort(rank(x))))[p:(p-9),]
```

Select Top 10 Features

Run Random Forest across imputed datasets

```

top10DataIndex      = matrix(as.numeric(sub('V','',topTenTypeOne)),nrow=10,ncol=100)
outrfTop10          = vector("list", 100)
yHatRFTop10         = matrix(0, nrow=100, ncol=200)
mseRFTop10          = rep(0,100)

for (i in 1:100) {
  xtrainT10          = trainDataSet[[i]][,c(top10DataIndex[,i],p+1)]
  xtestT10           = testDataSet[[i]][,top10DataIndex[,i]]
  ytest              = testDataSet[[i]][,p+1]
# Random Forest
  outrfTop10[[i]] = randomForest(y~., data = xtrainT10, ntree = 500, importance = FALSE, mtry
    = 10/3, na.action = na.roughfix)
# prediction
  yHatRFTop10[i,] = predict(outrfTop10[[i]], newdata = na.roughfix(xtestT10) , type = "response"
)
  mseRFTop10[i]    = mean((yHatRFTop10[,i]-ytest)^2)
}

```

MCC, RMSE and Confusion Matrix averaged over all 100 datasets

Calculate and capture resulting aggregated performance metrics.

```

#Calculate MCC - Evaluation of Variable Selection (Top 10 version)
vipTypeOne      = apply(as.data.frame(varImpTypeOne),1, function(x)names(sort(rank(x), decreasing = TRUE)))
vipTypeOneIndex = matrix(as.numeric(sub('V','',vipTypeOne)),nrow=p,ncol=100)
predic          = matrix(FALSE, nrow=p, ncol=100) # set up matrix
for (i in 1:100) {
  predic[vipTypeOneIndex[1:10,i],i] = TRUE # Top 10 VIP reset to TRUE
}
perfect         = c(rep(TRUE, times = 10), rep(FALSE, times = p-10))
mcc             = rep(0,100)
mccF2           = function(x) {mcc(preds = x, actuals = perfect)}
mcc             = apply(predic,2,mccF2)
avgMcc          = mean(mcc)

# Capture results data
truePosT10      = falsePosT10 = trueNegT10 = falseNegT10 = rep(0,p)
truePosT10      = apply(predic[1:10,],2,sum) # true positives
falsePosT10     = 10-truePosT10 # False Positives
falseNegT10     = apply(predic[11:p,],2,sum) # False Negatives
trueNegT10      = p-falseNegT10-falsePosT10-truePosT10

resultsMat      = data.frame(
  scenID      = scenID,
  scenName    = scenName,
  method      = "Top10",
  rmse        = round((sqrt(mean(unlist(mseRFTop10))))),2),
  avgMcc      = avgMcc,
  tp          = mean(truePosT10),
  tn          = mean(trueNegT10),
  fp          = mean(falsePosT10),
  fn          = mean(falseNegT10),
  seRMSE      = sd(sqrt(unlist(mseRFTop10))) / sqrt(length(unlist(mseRFTop10))),
  seMCC       = sd(unlist(mcc)) / sqrt(length(unlist(mcc))),
  seTP        = sd(truePosT10) / sqrt(length(truePosT10)),
  seTN        = sd(trueNegT10) / sqrt(length(trueNegT10)),
  seFP        = sd(falsePosT10) / sqrt(length(falsePosT10)),
  seFN        = sd(falseNegT10) / sqrt(length(falseNegT10))
)

```

Select Features with CV Threshold Method

As there isn't a CV Threshold algorithm for RF, the CV Threshold results are taken from MIRL for each of the 100 datasets.

Load threshold result from MIRL

```

# reset confusion matrix value temporary values
rmse = avgMCC = truePos = trueNeg = falsePos = falseNeg = 0
# Load MIRL threshold results
threshRes          = readRDS("S38MIRLThreshPred.Rds")

outrfThresh        = threshDataIndex = vector("list", 100)
yHatRFThresh       = matrix(0, nrow=100, ncol=200)  # n = 200
mseRFThresh        = rep(0,100)
predicThresh       = matrix(FALSE, nrow=p, ncol=100)

for (i in 1:100) {
  threshDataIndex[[i]] = vipTypeOneIndex[1:length(threshRes[[i]]),i]  # predictor indices in rows
  xtrainThresh         = trainDataSet[[i]][,c(threshDataIndex[[i]],p+1)]  # predictor indices in columns, y in last column
  xtestThresh          = testDataSet[[i]][,threshDataIndex[[i]]]
# calcs for MCC
  predicThresh[threshDataIndex[[i]],i] = TRUE  # VIP reset to TRUE for threshold selected features
# Random Forest
  outrfThresh[[i]] = randomForest(y~., data = xtrainThresh, ntree = 500, importance = FALSE, mtry = max(1,length(threshRes[[i]])/3), na.action = na.roughfix)
# prediction
  yHatRFThresh[i,] = predict(outrfThresh[[i]], newdata = na.roughfix(xtestThresh) , type = "response")
  mseRFThresh[i] = mean((yHatRFThresh[,i]-yTest[[i]])^2)
}

```

MCC, RMSE and Confusion Matrix averaged over all 100 datasets

Calculate and capture resulting aggregated performance metrics.

```

#Calculate MCC
mccThresh      = rep(0,10)
mcc            = apply(predicThresh,2,mccF2)
avgMcc        = round(mean(mcc),3)
# Capture results data
truePosThresh  = falsePosThresh = trueNegThresh = falseNegThresh = rep(0,p)
truePosThresh  = apply(predicThresh[1:10,],2,sum) # true positives
falsePosThresh = apply(predicThresh[11:p,],2,sum) # False Positives
falseNegThresh = 10-truePosThresh                # False Negatives
trueNegThresh  = p-truePosThresh-falsePosThresh-falseNegThresh

resultsMat = resultsMat %>% add_row(
  scenID    = scenID,
  scenName  = scenName,
  method    = "Threshold",
  rmse      = round((sqrt(mean(unlist(mseRFThresh))))),2),
  avgMcc    = avgMcc,
  tp        = mean(truePosThresh),
  tn        = mean(trueNegThresh),
  fp        = mean(falsePosThresh),
  fn        = mean(falseNegThresh),
  seRMSE    = sd(sqrt(unlist(mseRFThresh))) / sqrt(length(unlist(mseRFThresh))),
  seMCC     = sd(unlist(mcc)) / sqrt(length(unlist(mcc))),
  seTP      = sd(truePosThresh) / sqrt(length(truePosThresh)),
  seTN      = sd(trueNegThresh) / sqrt(length(trueNegThresh)),
  seFP      = sd(falsePosThresh) / sqrt(length(falsePosThresh)),
  seFN      = sd(falseNegThresh) / sqrt(length(falseNegThresh))
)

```

Results Summary

The results for both the Top 10 and CV Threshold scenarios using RF are shown below. For this scenario, predictive performance, as measured by RMSE, is similar between both scenarios. Feature selection performance is worse for the CV Threshold method, as one would expect given that the Top 10 scenario chooses exactly the right number of features. The CV Threshold scenario chooses an extra 6 features on average, across all 100 datasets.

```

kbl(resultsMat[,1:9], caption = "Table 1: Performance Metrics") %>% kable_styling(position="center", font_size = 12)

```

Table 1: Performance Metrics

| scenID | scenName | method | rmse | avgMcc | tp | tn | fp | fn |
|--------|----------|-----------|------|--------|------|-------|------|------|
| 38 | RF | Top10 | 1.56 | 0.622 | 6.85 | 46.85 | 3.15 | 3.15 |
| 38 | RF | Threshold | 1.56 | 0.517 | 7.33 | 41.45 | 8.55 | 2.67 |

Feature Selection Performance

Table 2 displays the number of times each truly associated feature (#1 through 10) is chosen over the 100 datasets. The features with the highest coefficients, #3-5 and 9-10 are chosen at least 90% of the time but there is a reduction in performance, when compared to the RL based methods, in selecting feature 8 as well as in selecting the features with the lower coefficient magnitudes. However, this method performs very well in selecting feature #6.

```
featureRes = data.frame(
  Weight      = as.factor(rep(1:5,2)/10),
  Association  = c(rep("Positive",5),rep("Negative",5)),
  Feature     = paste0('x', 1:10),
  Selected    = as.numeric(table(unlist(threshDataIndex))[1:10]))
featureResOrd = featureRes[order(featureRes$Weight), ][,c(2,4)]

testF = function(x) {
  ifelse(x<=25,"#FF7276",
    ifelse(x<=50,"#fed8b1",
      ifelse(x<=75,"lightyellow","lightgreen")))}

tblCol = testF(featureResOrd$Selected)

featureResOrd$Selected = color_bar(tblCol)(featureResOrd$Selected)

kbl(featureResOrd, escape = F, caption = "Table 2: Feature Selection Performance Metrics") %>%
  kable_paper("hover", full_width = F) %>%
  column_spec(3, width = "6cm") %>%
  group_rows("Coefficient: 0.1",1,2) %>% group_rows("Coefficient: 0.2",3,4) %>% group_rows("Coefficient: 0.3",5,6) %>%
  group_rows("Coefficient: 0.4",7,8) %>% group_rows("Coefficient: 0.5",9,10)
```

Table 2: Feature Selection Performance Metrics

| | Association | Selected |
|------------------|-------------|----------|
| Coefficient: 0.1 | | |
| 1 | Positive | 18 |
| 6 | Negative | 92 |
| Coefficient: 0.2 | | |
| 2 | Positive | 24 |
| 7 | Negative | 45 |
| Coefficient: 0.3 | | |
| 3 | Positive | 100 |
| 8 | Negative | 73 |
| Coefficient: 0.4 | | |
| 4 | Positive | 100 |
| 9 | Negative | 91 |
| Coefficient: 0.5 | | |
| 5 | Positive | 96 |
| 10 | Negative | 94 |

Turn off Parallelization

```
stopCluster(cl)
```

Save Workspace

```
save.image(file = paste0("S",scenID,scenName,".Rdata"))  
saveRDS(resultsMat, file = paste0("S",scenID,scenName,"Results.Rds"))  
saveRDS(threshDataIndex, file = paste0("S",scenID,scenName,"ThreshPred.Rds"))  
saveRDS(featureRes, file = paste0("S",scenID,scenName,"FeatureRes.Rds"))
```