Model 1: SVM

Junaira I. Ibrahim

2022-12-16

Importing Packages

```r
# Helper packages
library(dplyr)    # for data wrangling
```

```
## Warning: package 'dplyr' was built under R version 4.1.3

##
## 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
```

```r
library(ggplot2)  # for awesome graphics
```

```
## Warning: package 'ggplot2' was built under R version 4.1.3
```

```r
library(rsample)  # for data splitting
```

```
## Warning: package 'rsample' was built under R version 4.1.3
```

```r
library(caret)    # for classification and regression training
```

```
## Warning: package 'caret' was built under R version 4.1.3

## Loading required package: lattice
```

```r
library(kernlab)  # for fitting SVMs
```

```
## Warning: package 'kernlab' was built under R version 4.1.3

##
## Attaching package: 'kernlab'

## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```r
library(modeldata) #for Failure.binary data
```

```
## Warning: package 'modeldata' was built under R version 4.1.3
```

```r
library(forcats)
```

```
## Warning: package 'forcats' was built under R version 4.1.3
```

```
library(bestNormalize)
```

```
## Warning: package 'bestNormalize' was built under R version 4.1.3
```

```
library(pdp)      # for partial dependence plots, etc.
```

```
## Warning: package 'pdp' was built under R version 4.1.3
```

```
library(vip)      # for variable importance plots
```

```
## Warning: package 'vip' was built under R version 4.1.3
```

```
##
## Attaching package: 'vip'
```

```
## The following object is masked from 'package:utils':
##
##     vi
```

Importing the dataset The dataset used in this model is imported from `radiomics data`. It has 197 observations and 431 variables.

```
datard <- read.csv("D:/MS_STATISTICS/STT225 Statistical Computing/FINAL PROJECT/radiomics_completedata.c
dim(datard)
```

```
## [1] 197 431
```

####*Checking for null and missing values*

```
is.na(datard)
colSums(is.na(datard)) # no NA thus, there is no missing values
```

####*Checking for normality*

```
md1 = datard%>%select_if(is.numeric)
datamd1 = lapply(md1[,-1], shapiro.test)
r = lapply(datamd1, function(x)x$p.value) #Extracting p-value only
s=unlist(r)    #to convert a list to vector
sum(s[s>0.05])
```

```
## [1] 0.1350113
```

```
r$Entropy_cooc.W.ADC
```

```
## [1] 0.1350113
```

Based on the results, there is only one variable who is normally distributed (i.e. Entropy_cooc.W.ADC). All the rest are not normally distributed. Hence, we will try to normalize the data using `orderNorm()` function.

####*Normalizing the data*

```
datard_norm = datard[,c(3,5:length(names(datard)))]
datard_norm = apply(datard_norm,2,orderNorm)
datard_norm = lapply(datard_norm, function(x) x$x.t)   #to transformed original data
datard_norm = datard_norm%>%as.data.frame()
```

Check the new data for normality

```
datalr2 = lapply(datard_norm, shapiro.test)
r2 = lapply(datalr2, function(x) x$p.value)
s2 = unlist(r2)
sum(s2>0.05)
```

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

Based on the results, the rest of the variables is now normally distributed.

Substituting the normalized values into the original data, we have

```r
r3 = select(datard, c("Failure.binary",  "Entropy_cooc.W.ADC"))
datard_n = cbind(r3,datard_norm)
```

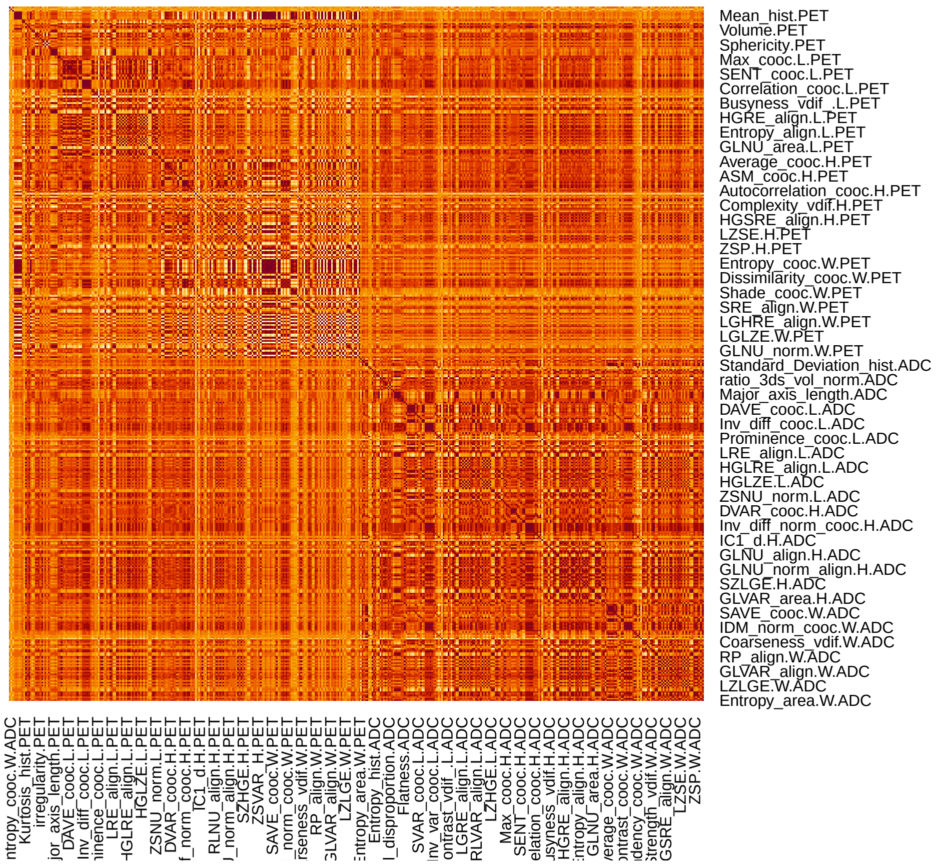
To set the `Failure.binary` into a factor level in dataset `datard_n`, we use the function `as.factor()` function.

```r
datard_n$Failure.binary=as.factor(datard_n$Failure.binary)
```

In this session, we will use `datard_n`.

Getting the correlation of the whole data

```r
#correlation
newdatard = select(datard_n, -c("Failure.binary"))
cor.newdatard = cor(newdatard)
corr = round(cor.newdatard,2) # 2 decimals
heatmap(corr,Rowv=NA,Colv=NA,scale="none",revC = T)
```



#Support Vector Machine Support vector machines (SVMs) offer a direct approach to binary classification.

SVMs use the kernel trick to enlarge the feature space using basis functions. A **Kernel Trick** is a simple method where a Non Linear data is projected onto a higher dimension space so as to make it easier to classify the data where it could be linearly divided by a plane. The popular kernel function used by SVMs are Linear `"svmLinear"`, Polynomial Kernel `"svmPoly"` and Radial basis kernel `"svmRadial"`.

In the following chunks, we use `getModelInfo()` function to extract the hyperparameters from various SVM implementations with different kernel functions.

```
# Linear (i.e., soft margin classifier)
caret::getModelInfo("svmLinear")$svmLinear$parameters
```

```
##   parameter   class label
## 1         C numeric  Cost
```

```
# Polynomial kernel
caret::getModelInfo("svmPoly")$svmPoly$parameters
```

```
##   parameter   class             label
## 1    degree numeric Polynomial Degree
## 2     scale numeric             Scale
## 3         C numeric              Cost
```

```
# Radial basis kernel
caret::getModelInfo("svmRadial")$svmRadial$parameters
```

```
##   parameter   class label
## 1     sigma numeric Sigma
## 2         C numeric  Cost
```

We can tune an SVM model with `train()`function with radial basis kernel using the data `rdtrain` and 10-fold CV.

The data `datard_n` is split into 80% of training data and 20% testing data.

```
#80% training data - 20% testing data
rdsplit <- initial_split(datard_n, prop = 0.8, strata = "Failure.binary")
rdsplit
```

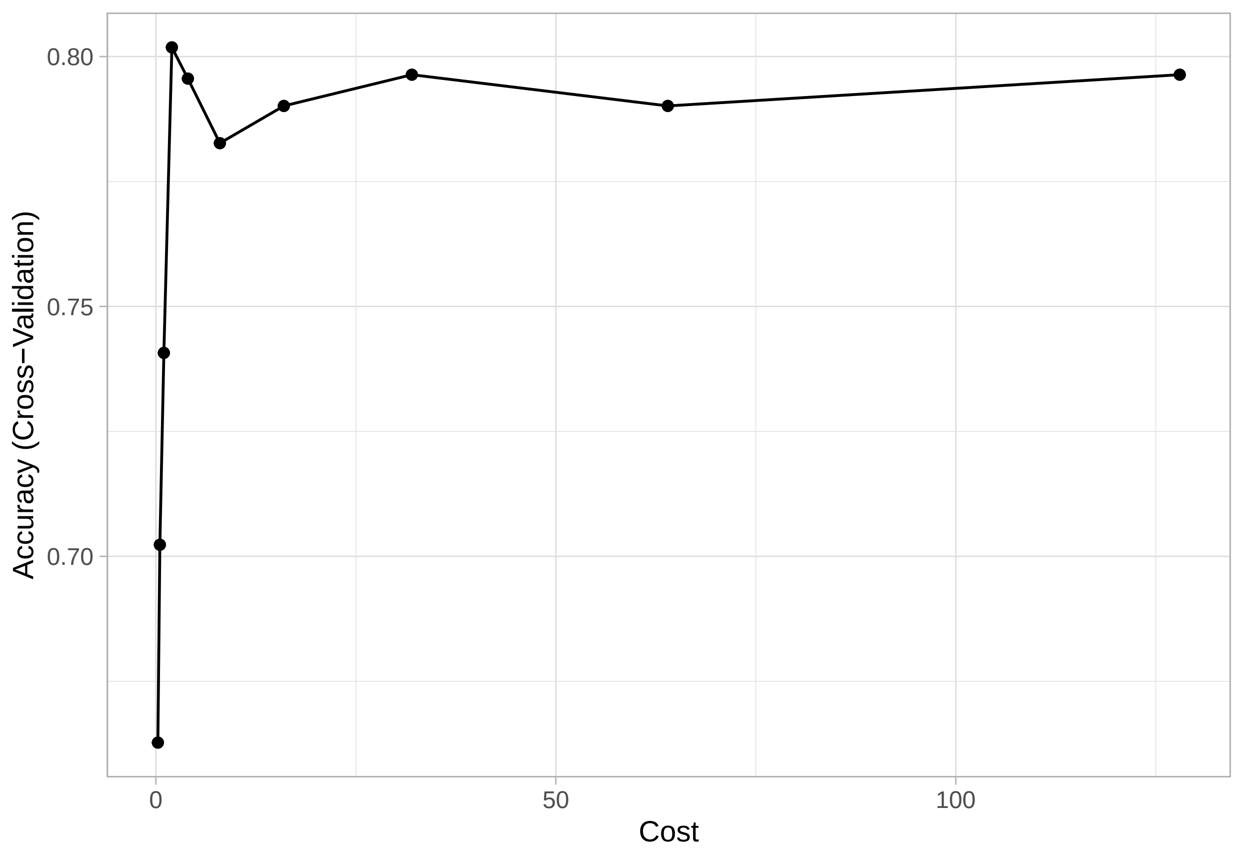
```
## <Training/Testing/Total>
## <157/40/197>
```

```
rdtrain <- training(rdsplit)
rdtest  <- testing(rdsplit)
```

```
set.seed(1854)  # for reproducibility
split_svm <- train(
  Failure.binary ~ .,
  data = rdtrain,
  method = "svmRadial",
  preProcess = c("center", "scale"),
  trControl = trainControl(method = "cv", number = 10),
  tuneLength = 10
)
```

Plotting the results, we see that smaller values of the cost parameter (C = 16-64) provide better cross-validated accuracy scores for these training data.

```
# Plot results
ggplot(split_svm) + theme_light()
```



```
# Print results
split_svm$results
```

```
##          sigma      C  Accuracy     Kappa AccuracySD   KappaSD
## 1  0.001838272   0.25 0.6627451 0.0000000 0.01891300 0.0000000
## 2  0.001838272   0.50 0.7023284 0.1640337 0.05605257 0.1606231
## 3  0.001838272   1.00 0.7407108 0.3543337 0.07178044 0.1596734
## 4  0.001838272   2.00 0.8018382 0.5274765 0.07671419 0.1924831
## 5  0.001838272   4.00 0.7955882 0.5235397 0.08028357 0.2025178
## 6  0.001838272   8.00 0.7826716 0.4985725 0.09350611 0.2356203
## 7  0.001838272  16.00 0.7901225 0.5162573 0.10016573 0.2493108
## 8  0.001838272  32.00 0.7963725 0.5359780 0.08950325 0.2136244
## 9  0.001838272  64.00 0.7901225 0.5191755 0.09042933 0.2158313
## 10 0.001838272 128.00 0.7963725 0.5359780 0.08950325 0.2136244
```

Control parameter

In order to obtain predicted class probabilities from an SVM, additional parameters need to be estimated. The predicted class probabilities are often more useful than the predicted class labels. For instance, we would need the predicted class probabilities if we were using an optimization metric like AUC. In that case, we can set `classProbs = TRUE` in the call to `trainControl()`.

```
class.weights = c("No" = 1, "Yes" = 10)

# Control params for SVM
ctrl <- trainControl(
  method = "cv",
  number = 10,
  classProbs = TRUE,
  summaryFunction = twoClassSummary
)

rdtrain$Failure.binary = fct_recode(rdtrain$Failure.binary,No="0",Yes="1")
```

Print the AUC values during Training

```
# Tune an SVM
set.seed(5628)  # for reproducibility
train_svm_auc <- train(
  Failure.binary ~ .,
  data = rdtrain,
  method = "svmRadial",
  preProcess = c("center", "scale"),
  metric = "ROC",
  trControl = ctrl,
  tuneLength = 10
)

# Print results
train_svm_auc$results
```

```
##          sigma      C       ROC      Sens      Spec      ROCSD     SensSD
## 1  0.001628158   0.25 0.7606061 0.8736364 0.4566667 0.12607169 0.09422732
## 2  0.001628158   0.50 0.7606061 0.8336364 0.4766667 0.12607169 0.17726884
## 3  0.001628158   1.00 0.7934545 0.8936364 0.4566667 0.12532974 0.07184439
## 4  0.001628158   2.00 0.8294545 0.9036364 0.5366667 0.09189053 0.06388120
## 5  0.001628158   4.00 0.8288182 0.8827273 0.5533333 0.09509364 0.12192593
## 6  0.001628158   8.00 0.8236364 0.8836364 0.5533333 0.09767431 0.12790607
## 7  0.001628158  16.00 0.8369697 0.8845455 0.5333333 0.10611360 0.09866417
## 8  0.001628158  32.00 0.8446364 0.8845455 0.5700000 0.11019957 0.09866417
## 9  0.001628158  64.00 0.8413030 0.8745455 0.6266667 0.11637329 0.12180160
## 10 0.001628158 128.00 0.8563030 0.9045455 0.6866667 0.11635923 0.08806146
##       SpecSD
## 1  0.2553381
## 2  0.2211362
## 3  0.2553381
## 4  0.2191172
## 5  0.2470567
## 6  0.2470567
## 7  0.2509242
## 8  0.2301100
## 9  0.2153378
## 10 0.1596292
```

```
confusionMatrix(train_svm_auc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   No  Yes
##        No  59.9 10.8
##        Yes  6.4 22.9
##
##  Accuracy (average) : 0.828
```

Based on the result, it is clear that we do a far better job at predicting the `Nos`.

Print the AUC values during Testing

```
rdtest$Failure.binary = fct_recode(rdtest$Failure.binary,No="0",Yes="1")

# Tune an SVM with radial
set.seed(5628)  # for reproducibility
test_svm_auc <- train(
  Failure.binary ~ .,
  data = rdtest,
  method = "svmRadial",
  preProcess = c("center", "scale"),
  metric = "ROC",  # area under ROC curve (AUC)
  trControl = ctrl,
  tuneLength = 10
)
```

```
# Print results
test_svm_auc$results
```

```
##          sigma      C       ROC      Sens Spec     ROCSD    SensSD    SpecSD
## 1  0.001420254   0.25 0.7833333 0.9333333 0.30 0.3147603 0.2108185 0.4216370
## 2  0.001420254   0.50 0.7833333 0.8000000 0.55 0.3147603 0.2698880 0.4972145
## 3  0.001420254   1.00 0.6833333 0.8166667 0.15 0.3884919 0.2539807 0.3374743
## 4  0.001420254   2.00 0.7833333 0.9000000 0.35 0.3147603 0.2249829 0.4116363
## 5  0.001420254   4.00 0.6583333 0.9000000 0.30 0.3736085 0.2249829 0.4216370
## 6  0.001420254   8.00 0.7833333 0.9000000 0.35 0.3147603 0.2249829 0.4743416
## 7  0.001420254  16.00 0.7833333 0.9333333 0.50 0.3147603 0.1405457 0.4714045
## 8  0.001420254  32.00 0.7833333 0.9000000 0.35 0.3147603 0.2249829 0.4743416
## 9  0.001420254  64.00 0.7833333 0.9000000 0.35 0.3147603 0.2249829 0.4743416
## 10 0.001420254 128.00 0.7833333 0.9000000 0.45 0.3147603 0.2249829 0.4972145
```

```
confusionMatrix(test_svm_auc)
```

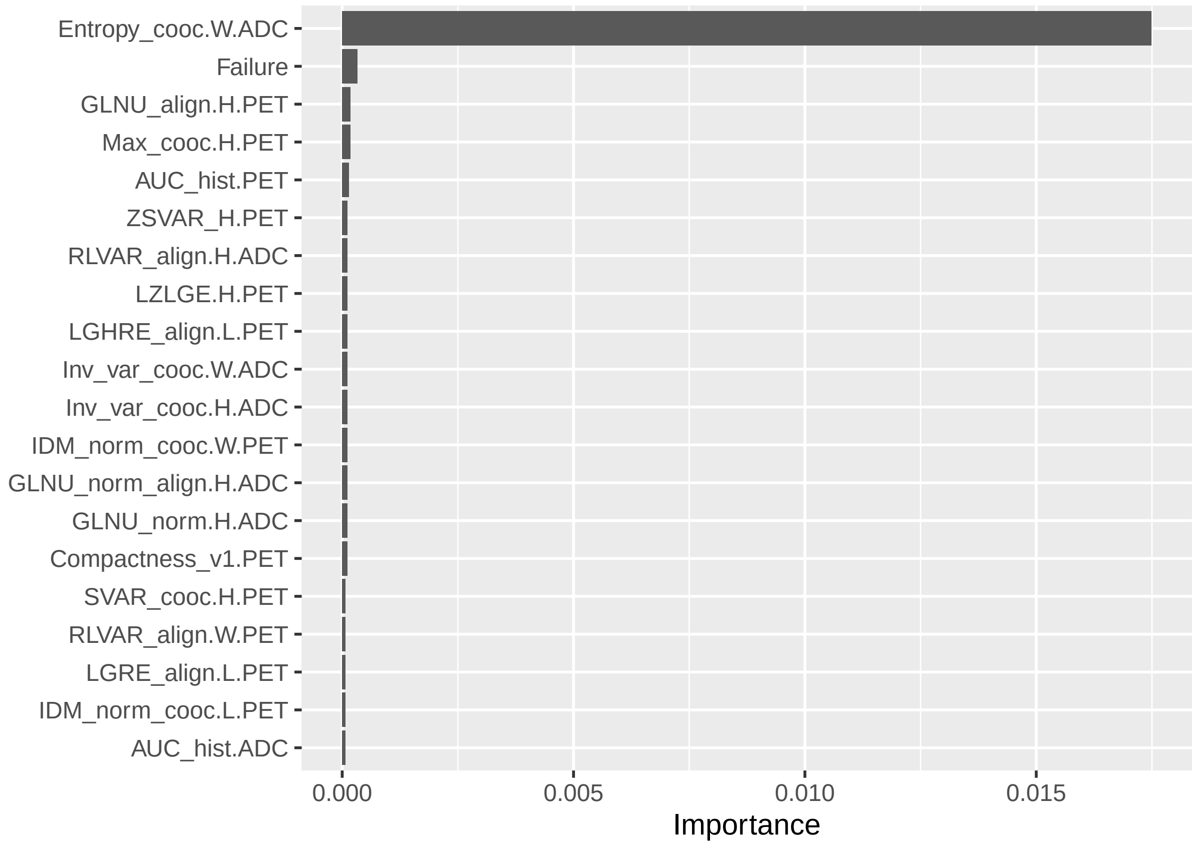
```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   No  Yes
##        No  60.0 22.5
##        Yes  5.0 12.5
##
##  Accuracy (average) : 0.725
```

Similar to training set, it is clear that we do a far better job at predicting the `Nos`.

To compute the vip scores we just call `vip()` with `method = "permute"` and pass our previously defined predictions wrapper to the `pred_wrapper` argument.

```
prob1 <- function(object, newdata) {
  predict(object, newdata = newdata, type = "prob")[, "Yes"]
}
```

```
# Variable importance plot
set.seed(2827)  # for reproducibility
vip(train_svm_auc, method = "permute", nsim = 5, train = rdtrain, num_features = 20, target = "Failure.
    pred_wrapper = prob1)
```

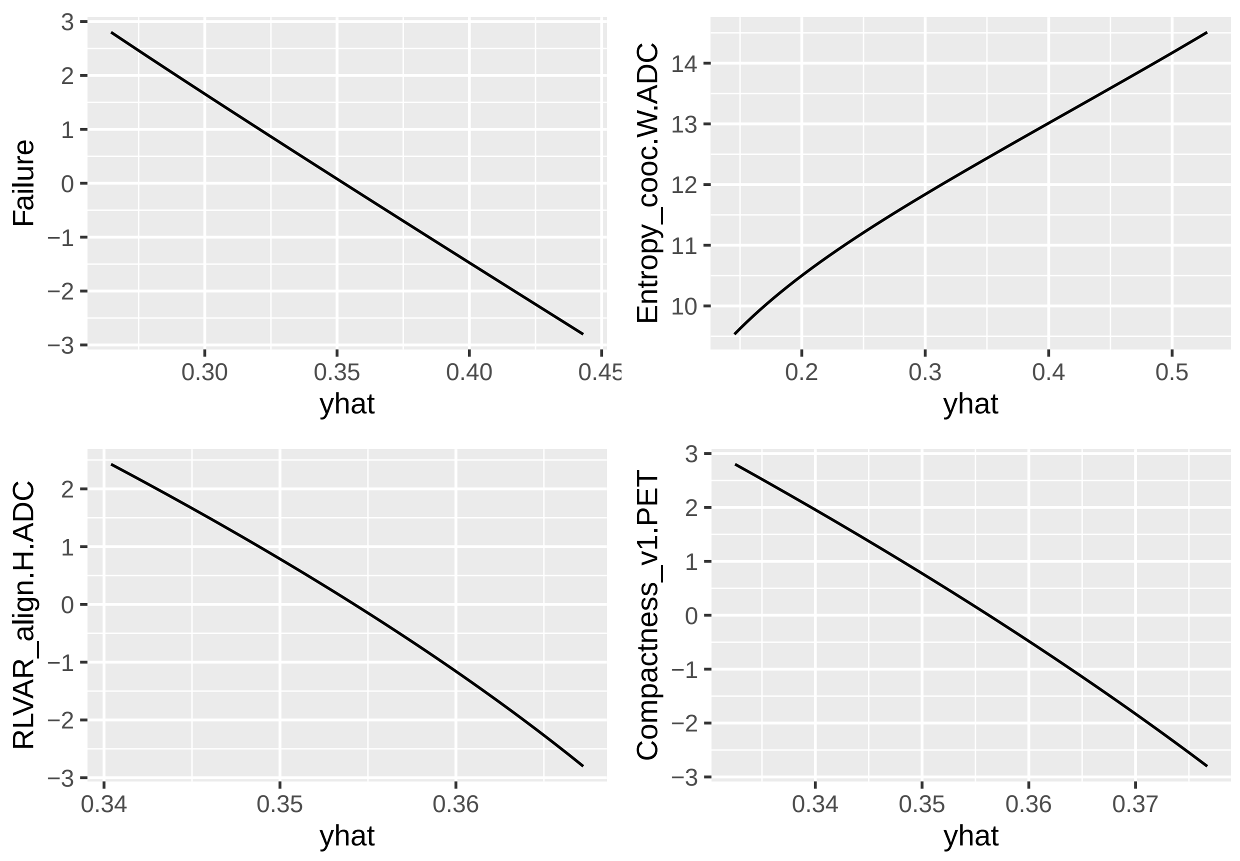


The results indicate that *Failure* and *Entropy_cooc.W.ADC* is the most important feature in predicting `Failure.binary`.

Next, we use the pdp package to construct PDPs for the top four features according to the permutation-based variable importance scores.

```
features1 <- c("Failure", "Entropy_cooc.W.ADC",
              "RLVAR_align.H.ADC", "Compactness_v1.PET")
pdps <- lapply(features1, function(x) {
  partial(train_svm_auc, pred.var = x, which.class = 2,
          prob = TRUE, plot = TRUE, plot.engine = "ggplot2") +
    coord_flip()
})

grid.arrange(grobs = pdps,  ncol = 2)
```



Failure
yhat
Entropy_cooc.W.ADC
yhat
RLVAR_align.H.ADC
yhat
Compactness_v1.PET
yhat