# Model 1

SITTY AZQUIA M. CAMAMA

2022-12-15

## Ensemble Classification Model

In this document, we will perform Ensemble Classification Model using **radiomics data**.

### Load Helper and Modeling Packages

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

```
##
## 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(ggplot2)     # for awesome plotting
library(doParallel)  # for parallel backend to foreach
```

```
## Loading required package: foreach
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
library(foreach)     # for parallel processing with for loops
library(rsample)     # for data splitting

# Modeling packages for Bagging
library(caret)       # for general model fitting
```

```
## Loading required package: lattice
```

```
library(rpart)       # for fitting decision trees
library(ipred)       # for fitting bagged decision trees
library(e1071)
```

```
##
## Attaching package: 'e1071'
```

```
## The following object is masked from 'package:rsample':
##
##     permutations
```

```
library(ROCR)
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
# Modeling packages for Random Forest
library(ranger)      # a c++ implementation of random forest
library(h2o)         # a java-based implementation of random forest
```

```
##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit https://docs.h2o.ai
##
## ----------------------------------------------------------------------
```

```
##
## Attaching package: 'h2o'
```

```
## The following object is masked from 'package:pROC':
##
##     var
```

```
## The following objects are masked from 'package:stats':
##
##     cor, sd, var
```

```
## The following objects are masked from 'package:base':
##
##     %*%, %in%, &&, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc
```

```
# Modeling packages for SVM
library(caret)    # for classification and regression training
library(kernlab)  # for fitting SVMs
```

```
##
## Attaching package: 'kernlab'
```

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

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

# Model interpretability packages
library(pdp)      # for partial dependence plots, etc.
library(vip)      # for variable importance plots
```

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

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

```
# Model for normalization
library(bestNormalize)
```

## Load Data Sets

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

```
radiomicsdata <- read.csv("C:/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
```

### Normalize data

To normalize 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 `rdnorm` variables.

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

## Splitting

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

```
# convert response column to a factor
ndata$Failure.binary=as.factor(ndata$Failure.binary)

set.seed(123) # make bootstrapping reproducible
split = initial_split(ndata,prop = 0.8 ,strata = "Failure.binary")
split_train <- training(split)
split_test  <- testing(split)
```

## 1. Bagging

This section provides an example of how to build an ensemble of predictions using bagging.Bagging is also known as bootstrap aggregating prediction models, is a general method for fitting multiple versions of a prediction model and then combining (or ensembling) them into an aggregated prediction and is designed to improve the stability and accuracy of regression and classification algorithms.

### Train Bagged Model

We can run the model by using `bagging()` function. We use `nbagg` to control how many iterations to include in the bagged model. As a general rule, the more trees the better. By using 100 nbagg, We have 0.1146 OOB error.

```
# make bootstrapping reproducible
set.seed(123)

# train bagged model
bmodel1 <- bagging(
  formula = Failure.binary ~ .,
  data = split_train,
  nbagg = 100,
  coob = TRUE,
  control = rpart.control(minsplit = 2, cp = 0)
)

bmodel1
```

```
##
## Bagging classification trees with 100 bootstrap replications
##
## Call: bagging.data.frame(formula = Failure.binary ~ ., data = split_train,
##     nbagg = 100, coob = TRUE, control = rpart.control(minsplit = 2,
##         cp = 0))
##
## Out-of-bag estimate of misclassification error:  0.1146
```

### Train the model using caret

We can also use bagging within `caret` and use `cv method` with 10-fold, to determine how effectively our ensemble will generalize. In this model, our accuracy is 0.8841176.

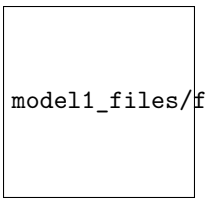
```
bmodel2 <- train(
  Failure.binary ~ .,
  data = split_train,
  method = "treebag",
  trControl = trainControl(method = "cv", number = 10),
  nbagg = 200,
  control = rpart.control(minsplit = 2, cp = 0)
)

bmodel2
```

```
## Bagged CART
##
## 157 samples
## 430 predictors
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 142, 141, 141, 141, 141, 142, ...
## Resampling results:
##
##   Accuracy   Kappa
```

```
##   0.8841176  0.7402472
```

### Print the AUC values during Training

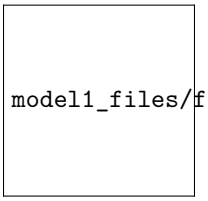
```
# Compute predicted probabilities on training data
prob.train <- predict(bmodel2, split_train, type = "prob")[,2]

# Compute AUC metrics for cv_model1,2 and 3
perf.train <- prediction(prob.train, split_train$Failure.binary) %>%
  performance(measure = "tpr", x.measure = "fpr")


# Plot ROC curves
plot(perf.train, col = "black", lty = 2)
```




```
# ROC plot for training data
roc( split_train$Failure.binary ~ prob.train, plot=TRUE, legacy.axes=FALSE,
    percent=TRUE, col="black", lwd=2, print.auc=TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

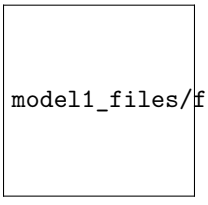



```
##
## Call:
## roc.formula(formula = split_train$Failure.binary ~ prob.train,     plot = TRUE, legacy.axes = FALSE,
##
## Data: prob.train in 104 controls (split_train$Failure.binary 0) < 53 cases (split_train$Failure.bina
## Area under the curve: 100%
```

### Print the Top 20 Important Features during Training

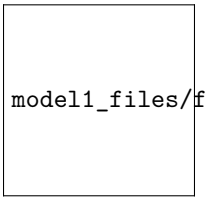
We use `vip()` function to construct a variable importance plot (VIP) of the top 20 features in the `bmodel2` model.

```
vip::vip(bmodel2, num_features = 20)
```



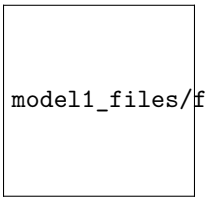
Partial dependence plots to understand the relationship between Failure.binary and the Entropy_cooc.W.ADC and Failure features. Partial dependence plots tell us visually how each feature influences the predicted output, on average.

```
# Construct partial dependence plots
p1 <- pdp::partial(
  bmodel2, pred.var = names(ndata)[3],
  grid.resolution = 20 ) %>%
  autoplot()

p2 <- pdp::partial(
  bmodel2, pred.var = names(ndata)[4],
  grid.resolution = 20) %>%
  autoplot()

gridExtra::grid.arrange(p1, p2, nrow = 1)
```




**Print the AUC values during Testing**

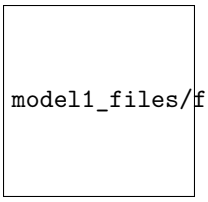
```
# Compute predicted probabilities on testing data
prob.test <- predict(bmodel2, split_test, type = "prob")[,2]

# Compute AUC metrics
perf.test <- prediction(prob.test, split_test$Failure.binary) %>%
  performance(measure = "tpr", x.measure = "fpr")


# Plot ROC curves
plot(perf.test, col = "black", lty = 2)
```




```
# ROC plot for testing data
roc( split_test$Failure.binary ~ prob.test, plot=TRUE, legacy.axes=FALSE,
    percent=TRUE, col="black", lwd=2, print.auc=TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```




```
##
## Call:
## roc.formula(formula = split_test$Failure.binary ~ prob.test,     plot = TRUE, legacy.axes = FALSE, p
##
## Data: prob.test in 26 controls (split_test$Failure.binary 0) < 14 cases (split_test$Failure.binary 1
## Area under the curve: 92.99%
```

## 2. Random Forest

This section provides an example of how to build Random Forests. Random forests are built using the same fundamental principles as bagging and decision trees.Random forests help to reduce tree correlation by injecting more randomness into the tree-growing process.

### Train Random Forest Model

Without any tuning, all hyperparameters set to their default values. Then, we get 0.347878 OOB RMSE.

```
# make bootstrapping reproducible
set.seed(123)

# number of features
n_features <- length(setdiff(names(split_train), "Failure.binary"))

# train a default random forest model
rf <- ranger(
  Failure.binary ~ .,
  data = split_train,
  mtry = floor(n_features / 3),
  respect.unordered.factors = "order",
  seed = 123
)

# get OOB RMSE
(default_rmse <- sqrt(rf$prediction.error)) #[1] 0.347878
```

```
## [1] 0.347878
```

### Create Hyperparameter Grid and Grid search

Despite the fact that random forests work well right out of the box, there are several tunable hyperparameters we should take into account when training a model. One way to become more strategic is to consider how we proceed through our grid search

```
# create hyperparameter grid
hyper_grid <- expand.grid(
  mtry = floor(n_features * c(.05, .15, .25, .333, .4)),
  min.node.size = c(1, 3, 5, 10),
  replace = c(TRUE, FALSE),
  sample.fraction = c(.5, .63, .8),
```

```
  rmse = NA
)

# execute full cartesian grid search
for(i in seq_len(nrow(hyper_grid))) {
  # fit model for ith hyperparameter combination
  fit <- ranger(
    formula         = Failure.binary ~ .,
    data            = split_train,
    num.trees       = n_features * 10,
    mtry            = hyper_grid$mtry[i],
    min.node.size   = hyper_grid$min.node.size[i],
    replace         = hyper_grid$replace[i],
    sample.fraction = hyper_grid$sample.fraction[i],
    verbose         = FALSE,
    seed            = 123,
    respect.unordered.factors = 'order',
  )
  # export OOB error
  hyper_grid$rmse[i] <- sqrt(fit$prediction.error)
}

# assess top 10 models
hyper_grid %>%
  arrange(rmse) %>%
  mutate(perc_gain = (default_rmse - rmse) / default_rmse * 100) %>%
  head(10)
```

```
##    mtry min.node.size replace sample.fraction      rmse perc_gain
## 1   172             1   FALSE            0.50 0.3290597  5.409470
## 2   172             3   FALSE            0.50 0.3290597  5.409470
## 3   172            10   FALSE            0.50 0.3290597  5.409470
## 4   143             5    TRUE            0.50 0.3385996  2.667147
## 5   143            10    TRUE            0.50 0.3385996  2.667147
## 6   143             1   FALSE            0.50 0.3385996  2.667147
## 7   172             5   FALSE            0.50 0.3385996  2.667147
## 8   107            10   FALSE            0.50 0.3385996  2.667147
## 9   143            10   FALSE            0.50 0.3385996  2.667147
## 10  143             1    TRUE            0.63 0.3385996  2.667147
```

### Convert Training data to h2o object

The following fits a default random forest model with `h2o` to illustrate that our baseline results (OOB RMSE= 0.3628804) are very similar to the baseline ranger model we fit earlier.

```
h2o.no_progress()
h2o.init(max_mem_size = "5g")
```

```
##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##     C:\Users\jobae\AppData\Local\Temp\RtmpC6v4Al\file9bc9c3cca/h2o_jobae_started_from_r.out
##     C:\Users\jobae\AppData\Local\Temp\RtmpC6v4Al\file9bc65985d39/h2o_jobae_started_from_r.err
##
```

```
##
## Starting H2O JVM and connecting:  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         2 seconds 748 milliseconds
##     H2O cluster timezone:       Asia/Taipei
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.38.0.1
##     H2O cluster version age:    2 months and 27 days
##     H2O cluster name:           H2O_started_from_R_jobae_osn291
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   5.00 GB
##     H2O cluster total cores:    4
##     H2O cluster allowed cores:  4
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.2.2 (2022-10-31 ucrt)
```

```
# convert training data to h2o object
train_h2o <- as.h2o(split_train)

# set the response column to Failure.binary
response <- "Failure.binary"

# set the predictor names
predictors <- setdiff(colnames(split_train), response)

h2o_rf1 <- h2o.randomForest(
  x = predictors,
  y = response,
  training_frame = train_h2o,
  ntrees = n_features * 10,
  seed = 123
)

h2o_rf1
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: drf
## Model ID:  DRF_model_R_1671186149809_1
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1            4300                     4300             1175498         3
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        13    6.93442          6         26    16.96977
##
##
## H2OBinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
```

```
##
## MSE:  0.1316822
## RMSE:  0.3628804
## LogLoss:  0.4184482
## Mean Per-Class Error:  0.1908563
## AUC:  0.886611
## AUCPR:  0.802224
## Gini:  0.7732221
## R^2:  0.4111331
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##         0  1    Error     Rate
## 0      80 24 0.230769  =24/104
## 1       8 45 0.150943    =8/53
## Totals 88 69 0.203822  =32/157
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold      value idx
## 1                        max f1  0.310247   0.737705  68
## 2                        max f2  0.268263   0.827703  83
## 3                  max f0point5  0.539567   0.801105  31
## 4                  max accuracy  0.539567   0.828025  31
## 5                 max precision  0.883501   1.000000   0
## 6                    max recall  0.144930   1.000000 116
## 7                max specificity  0.883501   1.000000   0
## 8               max absolute_mcc  0.539567   0.608451  31
## 9   max min_per_class_accuracy  0.370058   0.792453  60
## 10 max mean_per_class_accuracy  0.310247   0.809144  68
## 11                     max tns  0.883501 104.000000   0
## 12                     max fns  0.883501  52.000000   0
## 13                     max fps  0.030831 104.000000 156
## 14                     max tps  0.144930  53.000000 116
## 15                     max tnr  0.883501   1.000000   0
## 16                     max fnr  0.883501   0.981132   0
## 17                     max fpr  0.030831   1.000000 156
## 18                     max tpr  0.144930   1.000000 116
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

### Hyperparameter Grid for h2o

To execute a grid search in h2o we need our hyperparameter grid to be a list.

```
hyper_grid <- list(
  mtries = floor(n_features * c(.05, .15, .25, .333, .4)),
  min_rows = c(1, 3, 5, 10),
  max_depth = c(10, 20, 30),
  sample_rate = c(.55, .632, .70, .80)
)

# random grid search strategy
search_criteria <- list(
  strategy = "RandomDiscrete",
  stopping_metric = "mse",
  stopping_tolerance = 0.001,   # stop if improvement is < 0.1%
```

```
  stopping_rounds = 10,          # over the last 10 models
  max_runtime_secs = 60*5       # or stop search after 5 min.
)
```

**Perform grid search for h2o**

```
random_grid <- h2o.grid(
  algorithm = "randomForest",
  grid_id = "rf_random_grid",
  x = predictors,
  y = response,
  training_frame = train_h2o,
  hyper_params = hyper_grid,
  ntrees = n_features * 10,
  seed = 123,
  stopping_metric = "RMSE",
  stopping_rounds = 10,           # stop if last 10 trees added
  stopping_tolerance = 0.005,     # don't improve RMSE by 0.5%
  search_criteria = search_criteria
)
```

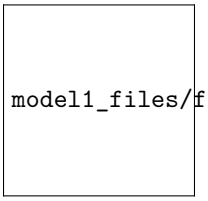
```
# collect the results and sort by our model performance metric
# of choice
random_grid_perf <- h2o.getGrid(
  grid_id = "rf_random_grid",
  sort_by = "mse",
  decreasing = FALSE
)
random_grid_perf
```

```
## H2O Grid Details
## ================
##
## Grid ID: rf_random_grid
## Used hyper parameters:
##   -  max_depth
##   -  min_rows
##   -  mtries
##   -  sample_rate
## Number of models: 240
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing mse
##    max_depth min_rows    mtries sample_rate                model_ids     mse
## 1   10.00000  3.00000 172.00000     0.63200 rf_random_grid_model_121 0.08195
## 2   20.00000  3.00000 172.00000     0.63200 rf_random_grid_model_162 0.08195
## 3   30.00000  3.00000 172.00000     0.63200 rf_random_grid_model_208 0.08195
## 4   10.00000  1.00000 172.00000     0.63200 rf_random_grid_model_216 0.08286
## 5   20.00000  1.00000 172.00000     0.63200  rf_random_grid_model_37 0.08286
##
## ---
##     max_depth min_rows   mtries sample_rate                model_ids     mse
## 235  10.00000 10.00000 21.00000     0.80000 rf_random_grid_model_114 0.15605
## 236  20.00000 10.00000 21.00000     0.80000 rf_random_grid_model_203 0.15605
```

```
## 237  30.00000 10.00000 21.00000     0.80000 rf_random_grid_model_234 0.15605
## 238  30.00000 10.00000 21.00000     0.55000 rf_random_grid_model_144 0.16302
## 239  10.00000 10.00000 21.00000     0.55000 rf_random_grid_model_240 0.16302
## 240  20.00000 10.00000 21.00000     0.55000  rf_random_grid_model_50 0.16302
```

### Print AUC values during Training

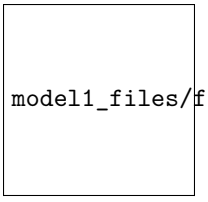
```
# Compute predicted probabilities on training data
prob.train1 <- predict(h2o_rf1, train_h2o, type = "prob")
prob.train1=as.data.frame(prob.train1)[,2]
train_h2o=as.data.frame(train_h2o)

# Compute AUC metrics for cv_model1,2 and 3
perf.train1 <- prediction(prob.train1,train_h2o$Failure.binary) %>%
  performance(measure = "tpr", x.measure = "fpr")


# Plot ROC curves
plot(perf.train1, col = "black", lty = 2)
```




```
# ROC plot for training data
roc( train_h2o$Failure.binary ~ prob.train1, plot=TRUE, legacy.axes=FALSE,
    percent=TRUE, col="black", lwd=2, print.auc=TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls > cases
```

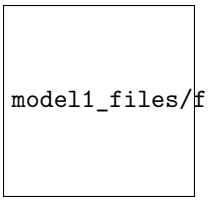



```
##
## Call:
## roc.formula(formula = train_h2o$Failure.binary ~ prob.train1,     plot = TRUE, legacy.axes = FALSE, p
##
## Data: prob.train1 in 104 controls (train_h2o$Failure.binary 0) > 53 cases (train_h2o$Failure.binary 1
## Area under the curve: 100%
```

### Print the Top 20 Important Features during Training

Using `h2o_rf1`, print the the top 20 important features in training data.

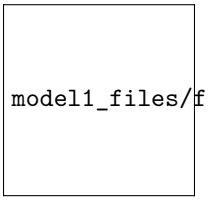
```
vip(h2o_rf1, num_features = 20)
```




The resulting VIPs shows the Top 20 most important variables based on impurity (left) and permutation (right).

```
# re-run model with impurity-based variable importance
rf_impurity <- ranger(
  formula = Failure.binary ~ .,
  data = split_train,
  num.trees = 2000,
  mtry = 32,
  min.node.size = 1,
  sample.fraction = .80,
  replace = FALSE,
  importance = "impurity",
  respect.unordered.factors = "order",
  verbose = FALSE,
  seed  = 123
)
```

```
# re-run model with permutation-based variable importance
rf_permutation <- ranger(
  formula = Failure.binary ~ .,
  data = split_train,
  num.trees = 2000,
  mtry = 32,
  min.node.size = 1,
  sample.fraction = .80,
  replace = FALSE,
  importance = "permutation",
  respect.unordered.factors = "order",
  verbose = FALSE,
  seed  = 123
)

#Plot the top importance for impurity and permutation
p1 <- vip::vip(rf_impurity, num_features = 20, bar = FALSE)
p2 <- vip::vip(rf_permutation, num_features = 20, bar = FALSE)

gridExtra::grid.arrange(p1, p2, nrow = 1)
```



## 3. 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
```

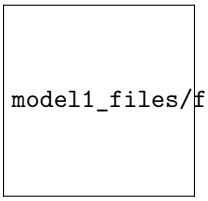
### Run SVM Model in Training phase

Using **split_train**, we can tune an SVM model with radial basis kernel.

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

Plot and print SVM model with with radial basis kernel.

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




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

```
##          sigma      C  Accuracy     Kappa AccuracySD   KappaSD
## 1  0.001998749   0.25 0.6627451 0.0000000 0.01891300 0.0000000
## 2  0.001998749   0.50 0.7378922 0.2715440 0.06418046 0.2198366
## 3  0.001998749   1.00 0.7779902 0.4565954 0.07142465 0.1608304
## 4  0.001998749   2.00 0.8023039 0.5196491 0.09057479 0.2186000
## 5  0.001998749   4.00 0.7889216 0.5030643 0.07639949 0.1942976
## 6  0.001998749   8.00 0.7697059 0.4653629 0.07092559 0.1830668
## 7  0.001998749  16.00 0.7763725 0.4861127 0.06283611 0.1498343
## 8  0.001998749  32.00 0.7826716 0.4985015 0.07602914 0.1806382
## 9  0.001998749  64.00 0.7960049 0.5248585 0.07147503 0.1670975
## 10 0.001998749 128.00 0.8018873 0.5429164 0.08701199 0.2010434
```

Control parameter

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

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

split_train$Failure.binary=fct_recode(split_train$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 = split_train,
  method = "svmRadial",
  preProcess = c("center", "scale"),
  metric = "ROC",  # area under ROC curve (AUC)
  trControl = ctrl,
  tuneLength = 10
)

# Print results
train_svm_auc$results
```

```
##          sigma      C       ROC      Sens      Spec      ROCSD     SensSD
## 1  0.001697891   0.25 0.8102727 0.8445455 0.5033333 0.09982583 0.12592723
## 2  0.001697891   0.50 0.8102727 0.8536364 0.5033333 0.09982583 0.12708861
## 3  0.001697891   1.00 0.8323939 0.8827273 0.5233333 0.09919217 0.11244425
## 4  0.001697891   2.00 0.8520606 0.9036364 0.6033333 0.09942461 0.09988055
## 5  0.001697891   4.00 0.8582121 0.9236364 0.6366667 0.09545946 0.09679909
## 6  0.001697891   8.00 0.8729697 0.9427273 0.5766667 0.11486557 0.06542227
## 7  0.001697891  16.00 0.8901818 0.9327273 0.6366667 0.13222606 0.07892762
## 8  0.001697891  32.00 0.8830000 0.9418182 0.5933333 0.13402578 0.06886193
## 9  0.001697891  64.00 0.8812121 0.9418182 0.6133333 0.15158268 0.05019704
## 10 0.001697891 128.00 0.8659697 0.9236364 0.6133333 0.15790577 0.08454491
##       SpecSD
```

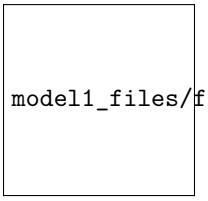
```
## 1  0.2224721
## 2  0.2224721
## 3  0.2403958
## 4  0.2157101
## 5  0.2235792
## 6  0.1937607
## 7  0.2027283
## 8  0.2968144
## 9  0.2563755
## 10 0.3182514
```

```
confusionMatrix(train_svm_auc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   No  Yes
##        No  61.8 12.1
##        Yes  4.5 21.7
##
##  Accuracy (average) : 0.8344
```

**Print the Top 20 important features during Training**

```
prob_yes <- 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 = split_train,
    num_features=20, target = "Failure.binary", metric = "auc",
    reference_class = "Yes", pred_wrapper = prob_yes)
```



model1_files/figure-latex/unnamed-chunk-33-1.pdf

**Print the AUC values during Testing**

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

# Tune an SVM with radial
set.seed(5628)  # for reproducibility
test_svm_auc <- train(
  Failure.binary ~ .,
  data = split_test,
  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  0.001959001   0.25 0.6750000 0.9666667    0 0.2872013 0.1054093      0
## 2  0.001959001   0.50 0.5750000 0.9333333    0 0.3320577 0.1405457      0
## 3  0.001959001   1.00 0.6250000 1.0000000    0 0.3148829 0.0000000      0
## 4  0.001959001   2.00 0.3083333 0.9000000    0 0.3168372 0.2249829      0
## 5  0.001959001   4.00 0.3500000 0.9000000    0 0.4021547 0.2249829      0
## 6  0.001959001   8.00 0.3916667 0.9000000    0 0.3889881 0.2249829      0
## 7  0.001959001  16.00 0.3083333 0.9000000    0 0.3514740 0.2249829      0
## 8  0.001959001  32.00 0.4250000 0.8333333    0 0.3976202 0.2832789      0
## 9  0.001959001  64.00 0.3750000 0.9333333    0 0.3833937 0.1405457      0
## 10 0.001959001 128.00 0.4083333 0.8666667    0 0.3937200 0.2810913      0
```

```
confusionMatrix(test_svm_auc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   No  Yes
##        No  62.5 35.0
##        Yes  2.5  0.0
##
##  Accuracy (average) : 0.625
```