

# Pokemon Project

## Data

— add description here

## Feature extraction

Import the df and visualize the columns' name.

```
df <- read.csv("data.csv")
names(df)
```

```
## [1] "X."          "Name"        "Type.1"      "Type.2"      "Total"
## [6] "HP"          "Attack"      "Defense"     "Sp..Atk"     "Sp..Def"
## [11] "Speed"       "Generation"  "Legendary"
```

Drop useless columns X., Name.

```
df <- df |>
  mutate(Legendary = as.integer(as.logical(Legendary))) |>
  select(-X., -Name)
```

Check that Tot is just a linear combination of other columns. If yes, drop it.

```
if (all((df$HP + df$Attack + df$Defense + df$SP..Attack + df$SP..Defense + df$Speed) == df$Total)) {
  df <- df |> select(-Total)
}
```

One-hot encoding from Type.1 and Type.2.

```
unique_types <- c(df$Type.1, df$Type.2) |> unique()

for (typ in unique_types) {
  if (typ == "") next
  df[typ] <- 0
  for (row in 1:nrow(df)) {
    has_type <- typ %in% df[row, c("Type.1", "Type.2")]
    if (has_type) {
      df[row, typ] <- 1
    }
  }
}
```

Encode Generation.

```
for (i in unique(df$Generation)) {
  # if (i == 1) next
  col_name <- paste0("gen_", i)
  df[col_name] <- 0
}

for (row in 1:nrow(df)) {
```

```

gen <- df[row, "Generation"]
# if (gen == 1) next
col_name <- paste0("gen_", gen)
df[row, col_name] <- 1
}

```

Drop the categorical columns that we don't need anymore and set Legendary to factor.

```

df <- df |>
  select(-Type.1, -Type.2, -Generation) |>
  mutate(Legendary = as.factor(ifelse(Legendary == 0, "No", "Yes")))

colnames(df)[c(1, 4, 5)] <- c("Hit_Point", "Special_Attack", "Special_Defense")

set.seed(123)
train_test_split <- function(df, perc_train = 0.7) {
  i_train <- sample(1:nrow(df), floor(0.7 * nrow(df)), F)
  list_out <- list(train = df[i_train, ], test = df[-i_train, ])
  return(list_out)
}
df_split <- train_test_split(df)

train <- df_split$train
test <- df_split$test

numerical_vars <- names(train)[1:6]

```

## Categorical data

We start by checking the frequency of Characteristics across our Pokemon population in the training sample:

```

train %>%
  select(-numerical_vars, -Legendary, -gen_1, -gen_2, -gen_3, -gen_4, -gen_5, -gen_6) %>%
  summarise(across(everything(), sum, na.rm = TRUE)) %>%
  pivot_longer(cols = everything(), names_to = "Feature", values_to = "Total_Sum") %>%
  mutate(Percentage = (Total_Sum / sum(Total_Sum)) * 100) %>%
  select(Feature, Percentage) %>%
  kbl()

```

Feature	Percentage
Grass	7.420495
Fire	5.418139
Water	11.660777
Bug	6.007067
Normal	7.891637
Poison	4.711425
Electric	4.240283
Ground	6.007067
Fairy	3.415783
Fighting	4.358068
Psychic	7.302709
Rock	4.711425
Ghost	3.651355
Ice	3.062426
Dragon	3.769140
Dark	3.886926
Steel	4.122497
Flying	8.362780

Check if the probability of having a Legendary is equal accross generation to decide whether to keep the variable.

```
train %>%
  select(gen_1, gen_2, gen_3, gen_4, gen_5, gen_6) %>%
  summarise(across(everything(), sum, na.rm = TRUE)) %>%
  pivot_longer(cols = everything(), names_to = "Feature", values_to = "Sum") %>%
  mutate(Percentage = (Sum / nrow(train)) * 100) %>% # Use nrow(train) here
  select(Feature, Percentage) %>%
  kbl()
```

Feature	Percentage
gen_1	20.357143
gen_2	13.750000
gen_3	20.535714
gen_4	14.464286
gen_5	21.250000
gen_6	9.642857

Now we can drop Generation 1, so that it is the baseline:

```
train <- train %>%
  select(-gen_1)
```

Since it's not equally likely to find a legendary Pokemon in each generation, with odd generations presenting more datapoints, it is important to keep it.

## Numerical data

```
# Numerical Variables
train %>%
  select(numerical_vars) %>%
  summary() %>%
  kbl()
```

	Hit_Point	Attack	Defense	Special_Attack	Special_Defense	Speed
	Min. : 1.00	Min. : 5.00	Min. : 5.00	Min. : 10.00	Min. : 20.00	Min. : 10.00
	1st Qu.: 54.00	1st Qu.: 55.00	1st Qu.: 50.00	1st Qu.: 50.00	1st Qu.: 53.00	1st Qu.: 45.75
	Median : 65.50	Median : 75.00	Median : 70.00	Median : 65.00	Median : 70.00	Median : 65.00
	Mean : 70.39	Mean : 79.29	Mean : 73.81	Mean : 72.60	Mean : 71.79	Mean : 68.59
	3rd Qu.: 84.00	3rd Qu.:100.00	3rd Qu.: 90.00	3rd Qu.: 94.25	3rd Qu.: 87.00	3rd Qu.: 90.00
	Max. :255.00	Max. :190.00	Max. :230.00	Max. :194.00	Max. :200.00	Max. :160.00

```
par(mfrow = c(2, 4), mar = c(3, 3, 3, 3))
lapply(numerical_vars, function(col_name) {
  hist(train[[col_name]], main = paste("Histogram of", col_name), xlab = "variable")
})
```

```
## [[1]]
## $breaks
## [1] 0 20 40 60 80 100 120 140 160 180 200 220 240 260
##
## $counts
## [1] 5 51 175 182 98 32 6 6 2 1 0 0 2
##
## $density
## [1] 4.464286e-04 4.553571e-03 1.562500e-02 1.625000e-02 8.750000e-03
## [6] 2.857143e-03 5.357143e-04 5.357143e-04 1.785714e-04 8.928571e-05
## [11] 0.000000e+00 0.000000e+00 1.785714e-04
##
## $mids
## [1] 10 30 50 70 90 110 130 150 170 190 210 230 250
##
## $xname
## [1] "train[[col_name]]"
##
## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
##
## [[2]]
## $breaks
## [1] 0 20 40 60 80 100 120 140 160 180 200
##
## $counts
## [1] 11 49 119 146 112 58 43 16 5 1
##
## $density
## [1] 9.821429e-04 4.375000e-03 1.062500e-02 1.303571e-02 1.000000e-02
## [6] 5.178571e-03 3.839286e-03 1.428571e-03 4.464286e-04 8.928571e-05
##
## $mids
## [1] 10 30 50 70 90 110 130 150 170 190
##
## $xname
## [1] "train[[col_name]]"
##
```

```

## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
##
## [[3]]
## $breaks
## [1] 0 20 40 60 80 100 120 140 160 180 200 220 240
##
## $counts
## [1] 11 57 149 154 104 49 22 7 2 3 0 2
##
## $density
## [1] 0.0009821429 0.0050892857 0.0133035714 0.0137500000 0.0092857143
## [6] 0.0043750000 0.0019642857 0.0006250000 0.0001785714 0.0002678571
## [11] 0.0000000000 0.0001785714
##
## $mids
## [1] 10 30 50 70 90 110 130 150 170 190 210 230
##
## $xname
## [1] "train[[col_name]]"
##
## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
##
## [[4]]
## $breaks
## [1] 0 20 40 60 80 100 120 140 160 180 200
##
## $counts
## [1] 10 84 153 128 88 47 29 14 6 1
##
## $density
## [1] 8.928571e-04 7.500000e-03 1.366071e-02 1.142857e-02 7.857143e-03
## [6] 4.196429e-03 2.589286e-03 1.250000e-03 5.357143e-04 8.928571e-05
##
## $mids
## [1] 10 30 50 70 90 110 130 150 170 190
##
## $xname
## [1] "train[[col_name]]"
##
## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
##
## [[5]]

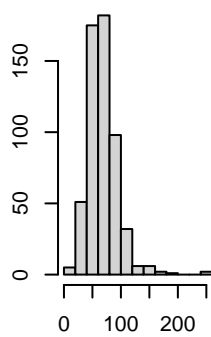
```

```

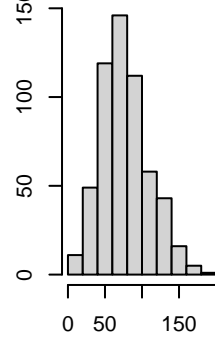
## $breaks
## [1] 20 40 60 80 100 120 140 160 180 200
##
## $counts
## [1] 59 159 169 103 52 10 7 0 1
##
## $density
## [1] 5.267857e-03 1.419643e-02 1.508929e-02 9.196429e-03 4.642857e-03
## [6] 8.928571e-04 6.250000e-04 0.000000e+00 8.928571e-05
##
## $mids
## [1] 30 50 70 90 110 130 150 170 190
##
## $xname
## [1] "train[[col_name]]"
##
## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
##
## [[6]]
## $breaks
## [1] 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
##
## $counts
## [1] 18 42 49 72 73 70 51 59 50 36 18 10 4 7 1
##
## $density
## [1] 0.0032142857 0.0075000000 0.0087500000 0.0128571429 0.0130357143
## [6] 0.0125000000 0.0091071429 0.0105357143 0.0089285714 0.0064285714
## [11] 0.0032142857 0.0017857143 0.0007142857 0.0012500000 0.0001785714
##
## $mids
## [1] 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155
##
## $xname
## [1] "train[[col_name]]"
##
## $equidist
## [1] TRUE
##
## attr("class")
## [1] "histogram"
par(mfrow = c(1, 1))

```

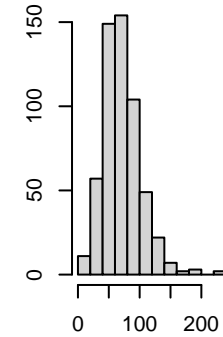
Histogram of Hit\_Point



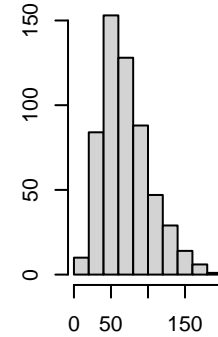
Histogram of Attack



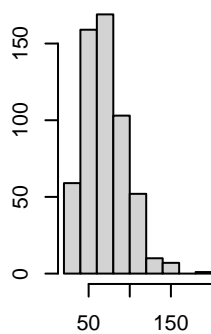
Histogram of Defense



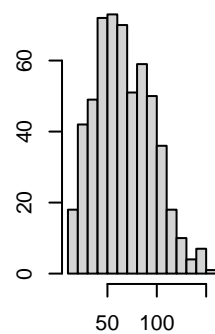
Histogram of Special\_Attack



Histogram of Special\_Defense



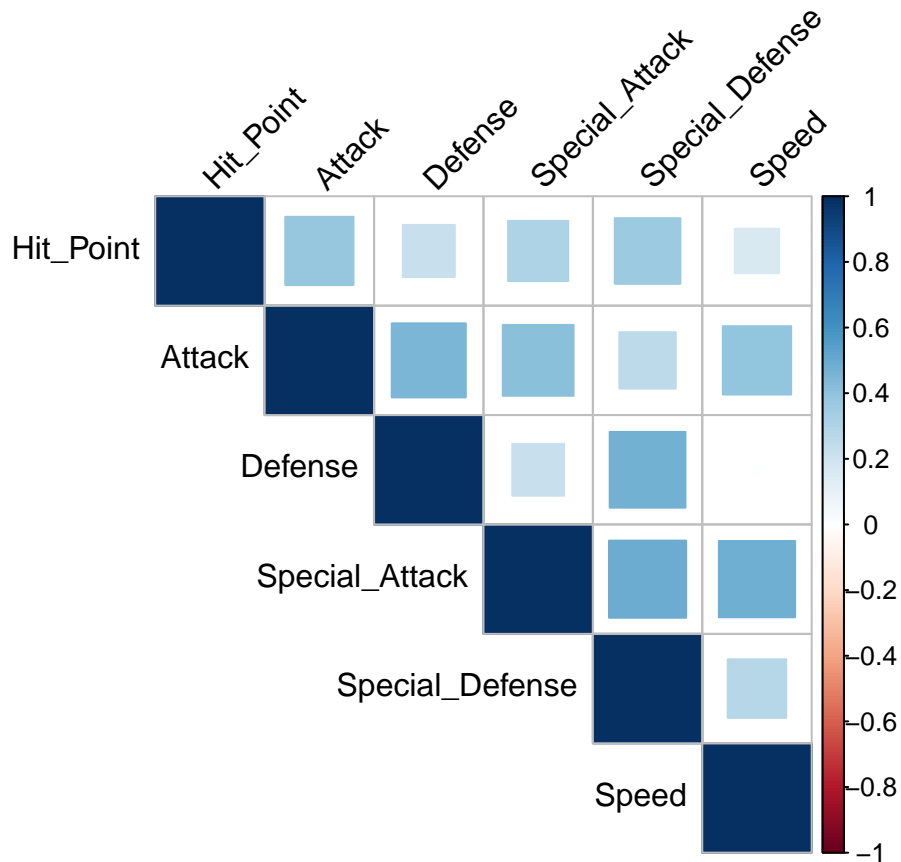
Histogram of Speed



```
cor_mat <- cor(train[numerical_vars])
print(cor_mat)
```

```
##           Hit_Point   Attack   Defense Special_Attack Special_Defense
## Hit_Point      1.0000000 0.3890117 0.22626454      0.3025508      0.3600523
## Attack         0.3890117 1.0000000 0.45833086      0.4189784      0.2655658
## Defense        0.2262645 0.4583309 1.00000000      0.2245206      0.4795059
## Special_Attack 0.3025508 0.4189784 0.22452059      1.0000000      0.4970868
## Special_Defense 0.3600523 0.2655658 0.47950586      0.4970868      1.0000000
## Speed          0.1647429 0.3902713 0.00647584      0.4858104      0.2862043
##
##           Speed
## Hit_Point      0.16474292
## Attack         0.39027129
## Defense        0.00647584
## Special_Attack 0.48581040
## Special_Defense 0.28620434
## Speed          1.00000000
```

```
corrplot(cor_mat, method = "square", type = "upper", tl.col = "black", tl.srt = 45)
```



```
scaling_params <- sapply(train[numerical_vars], mean)
scaling_params_sd <- sapply(train[numerical_vars], sd)

train_std <- train
train_std[numerical_vars] <- sweep(train_std[numerical_vars], 2, scaling_params, "-")
train_std[numerical_vars] <- sweep(train_std[numerical_vars], 2, scaling_params_sd, "/")

# Apply the same scaling to test data
test_std <- test
test_std[numerical_vars] <- sweep(test_std[numerical_vars], 2, scaling_params, "-")
test_std[numerical_vars] <- sweep(test_std[numerical_vars], 2, scaling_params_sd, "/")
```

Checks

```
sapply(list(mean = mean, sd = sd), mapply, train_std |> select(numerical_vars))
```

```
##              mean sd
## Hit_Point    1.462889e-16 1
## Attack      -4.257643e-17 1
## Defense     -2.249584e-16 1
## Special_Attack 2.896194e-17 1
## Special_Defense 2.191568e-16 1
## Speed       4.330444e-17 1
```

```
sapply(list(mean = mean, sd = sd), mapply, test_std |> select(numerical_vars))
```

```
##              mean      sd
## Hit_Point    -0.142726299 0.8682415
```



```

## Attack          -0.029657333 0.9690458
## Defense         0.003810264 1.0290146
## Special_Attack  0.022203552 1.0254181
## Special_Defense 0.014580900 1.1861049
## Speed          -0.036205070 1.0046306

cv_seed <- list(
  c(7, 18, 21, 34, 50, 67, 71, 85, 90, 44, 62, 37, 12, 55, 28, 99, 46, 19, 38, 68, 23),
  c(9, 16, 11, 40, 51, 53, 45, 64, 39, 57, 69, 27, 72, 61, 36, 56, 75, 80, 66, 26, 32),
  c(18, 25, 48, 31, 42, 35, 63, 22, 20, 77, 24, 74, 49, 10, 16, 82, 33, 13, 14, 58, 60),
  c(39, 41, 17, 55, 59, 26, 65, 30, 79, 19, 73, 12, 27, 70, 50, 84, 76, 28, 20, 35, 37),
  c(61, 43, 33, 44, 52, 72, 31, 78, 57, 49, 22, 76, 56, 47, 35, 69, 66, 21, 62, 9, 36),
  c(24, 83, 75, 59, 32, 64, 60, 52, 25, 58, 48, 71, 40, 50, 54, 39, 53, 23, 15, 12, 20),
  c(45, 14, 37, 19, 80, 53, 28, 55, 41, 23, 51, 29, 64, 47, 67, 60, 22, 32, 49, 66, 68),
  c(63, 15, 48, 40, 26, 34, 77, 39, 61, 29, 52, 46, 69, 73, 16, 59, 79, 41, 17, 10, 54),
  c(62, 55, 77, 56, 24, 38, 81, 22, 18, 71, 48, 63, 60, 35, 45, 73, 49, 68, 32, 50, 28),
  c(84, 36, 29, 68, 16, 59, 14, 79, 25, 57, 71, 34, 53, 67, 40, 51, 15, 46, 69, 76, 33),
  99 # Last element with a single integer
)

extract_best_f <- function(fit) fit$results[which.max(fit$results$F), ]

grid_search_threshold <- function(fit) {
  best_given_threshold <- data.frame(matrix(ncol = 4, nrow = 0))
  colnames(best_given_threshold) <- c("alpha", "lambda", "prob_threshold", "F1")
  all_thresholds <- seq(0.3, 0.8, 0.1)
  for (tr in all_thresholds) {
    res <- thresholder(fit, tr, F, "F1")
    best <- res[which.max(res$F1), ]
    best_given_threshold <- rbind(best_given_threshold, best)
  }
  return(best_given_threshold)
}

extract_best_threshold_f <- function(grid_df) {
  grid_df[which.max(grid_df$F1), ]
}

```

Fitting a logistic elastic net. Grid search of optimal alpha and lambda. CV. Maximize F.

```

grid_ <- expand.grid(
  .alpha = seq(0, 1, by = 0.1),
  .lambda = 10^(-c(0, 1, 10, 100, 1000))
)

train_control_1 <- trainControl(
  method = "cv",
  number = 10,
  classProbs = TRUE,
  summaryFunction = prSummary,
  savePredictions = "all",
  seeds = cv_seed
)

logistic_elnet_1 <- train(
  Legendary ~ .,
  data = train_std,

```

```

method = "glmnet",
family = "binomial",
metric = "F",
tuneGrid = grid_,
trControl = train_control_1,
intercept = FALSE
)

print(logistic_elnet_1$bestTune)

##      alpha lambda
## 13      0.2 1e-10

print(extract_best_f(logistic_elnet_1))

##      alpha lambda      AUC Precision      Recall      F      AUCSD PrecisionSD
## 11      0.2      0 0.9738336 0.9581359 0.9651207 0.9613257 0.00643858 0.02575424
##      RecallSD      FSD
## 11 0.02177563 0.01621389

best_f_1 <- grid_search_threshold(logistic_elnet_1)
print(best_f_1)

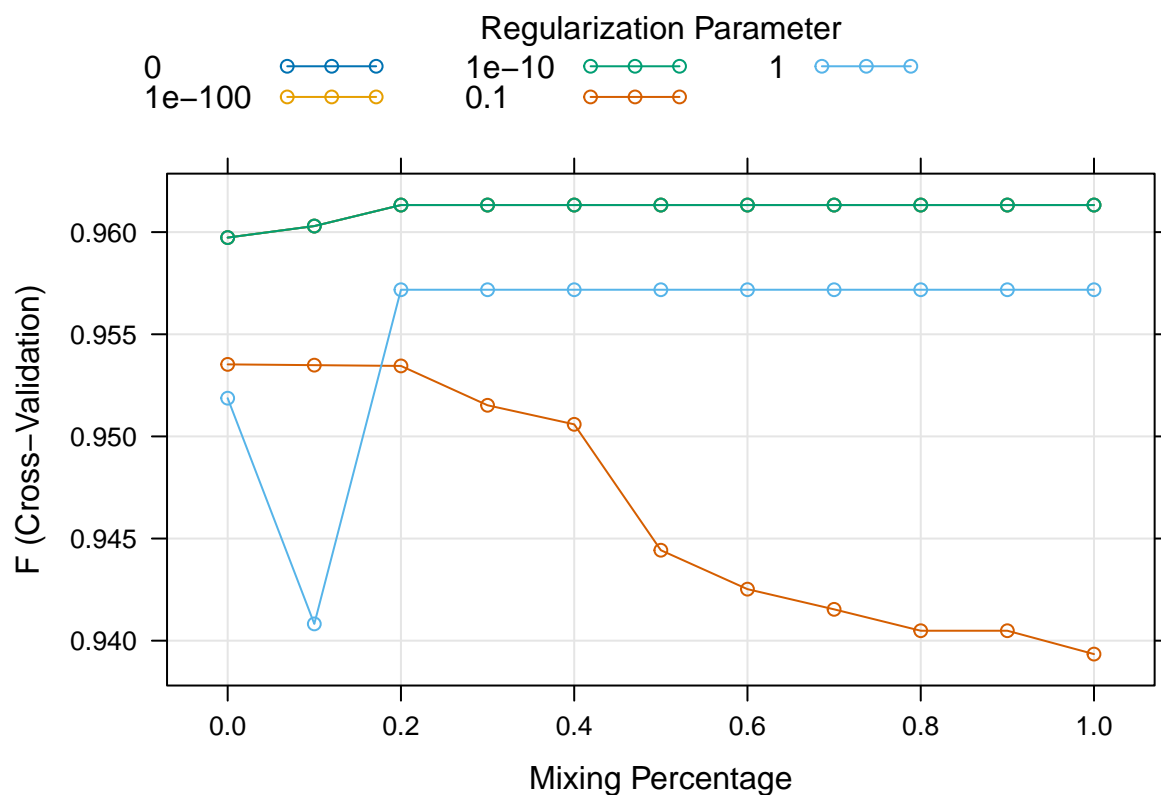
##      alpha lambda prob_threshold      F1
## 6      0.1      0      0.3 0.9617127
## 61     0.1      0      0.4 0.9604477
## 11     0.2      0      0.5 0.9613257
## 1      0.0      0      0.6 0.9639693
## 111    0.2      0      0.7 0.9536440
## 112    0.2      0      0.8 0.9480063

print(extract_best_threshold_f(best_f_1))

##      alpha lambda prob_threshold      F1
## 1      0      0      0.6 0.9639693

plot(logistic_elnet_1)

```



Fitting a logistic elastic net. Grid search of optimal alpha and lambda. SMOTE CV. Maximize F.

```
train_control_2 <- trainControl(
  method = "cv",
  number = 10,
  sampling = "smote",
  classProbs = TRUE,
  summaryFunction = prSummary,
  savePredictions = "all",
  seed = cv_seed
)
```

```
logistic_elnet_2 <- train(
  Legendary ~ .,
  data = train_std,
  method = "glmnet",
  family = "binomial",
  metric = "F",
  tuneGrid = grid_,
  trControl = train_control_2,
  intercept = FALSE
)
print(logistic_elnet_2$bestTune)
```

```
## alpha lambda
## 25 0.4 1
```

```
print(extract_best_f(logistic_elnet_2))
```

```
##      alpha lambda AUC Precision Recall      F AUCSD PrecisionSD RecallSD
## 25    0.4      1  0 0.9179107      1 0.9571795    0 0.008648344      0
##      FSD
## 25 0.004694285
```

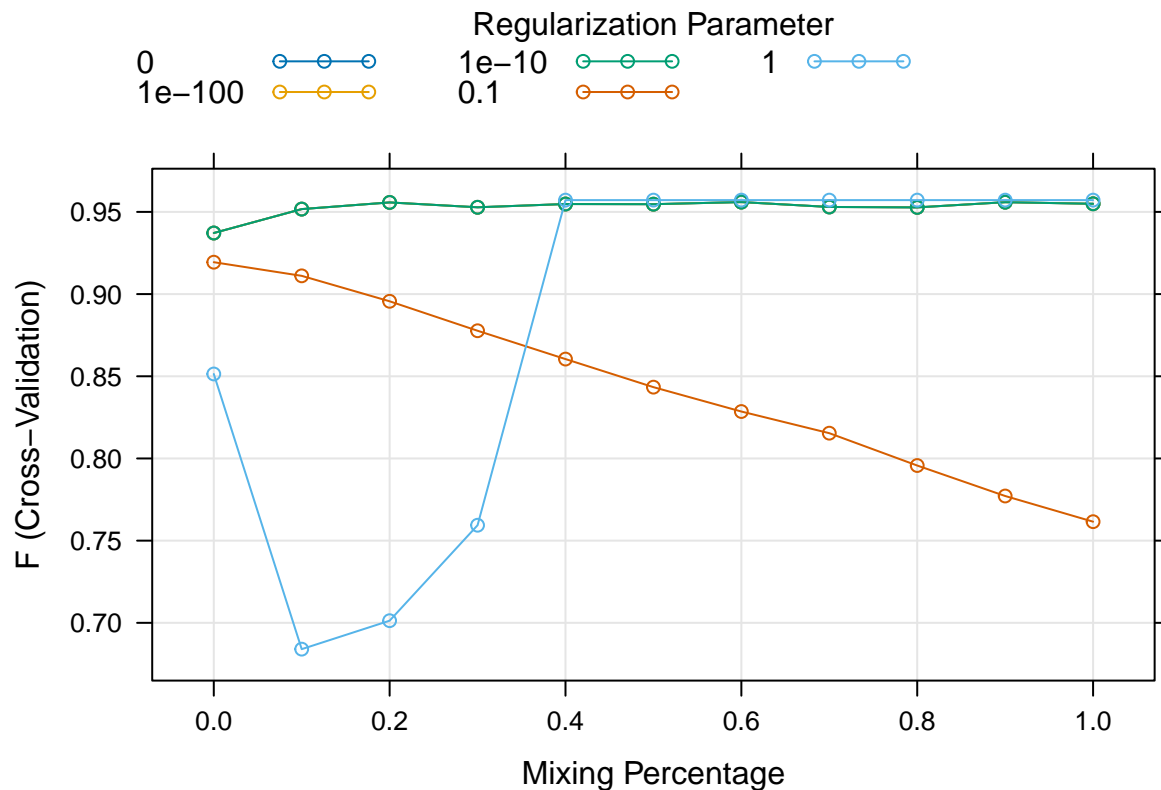
```
best_f_2 <- grid_search_threshold(logistic_elnet_2)
print(best_f_2)
```

```
##      alpha lambda prob_threshold      F1
## 4      0.0      0.1      0.3 0.9662289
## 15     0.2      1.0      0.4 0.9653466
## 31     0.6      0.0      0.5 0.9559835
## 41     0.8      0.0      0.6 0.9514466
## 36     0.7      0.0      0.7 0.9472411
## 51     1.0      0.0      0.8 0.9350926
```

```
print(extract_best_threshold_f(best_f_2))
```

```
##      alpha lambda prob_threshold      F1
## 4      0      0.1      0.3 0.9662289
```

```
plot(logistic_elnet_2)
```



Fitting a logistic elastic net. Grid search of optimal alpha and lambda. Upsampling CV. Maximize F.

```

train_control_3 <- trainControl(
  method = "cv",
  number = 10,
  sampling = "up",
  classProbs = TRUE,
  summaryFunction = prSummary,
  savePredictions = "all",
  seed = cv_seed
)

logistic_elnet_3 <- train(
  Legendary ~ .,
  data = train_std,
  method = "glmnet",
  family = "binomial",
  metric = "F",
  tuneGrid = grid_,
  trControl = train_control_3,
  intercept = FALSE
)
print(logistic_elnet_3$bestTune)

##      alpha lambda
## 25    0.4      1

print(extract_best_f(logistic_elnet_3))

##      alpha lambda AUC Precision Recall      F AUCSD PrecisionSD RecallSD
## 25    0.4      1    0 0.9179375      1 0.9571954      0 0.008346938      0
##      FSD
## 25 0.00453091

best_f_3 <- grid_search_threshold(logistic_elnet_3)
print(best_f_3)

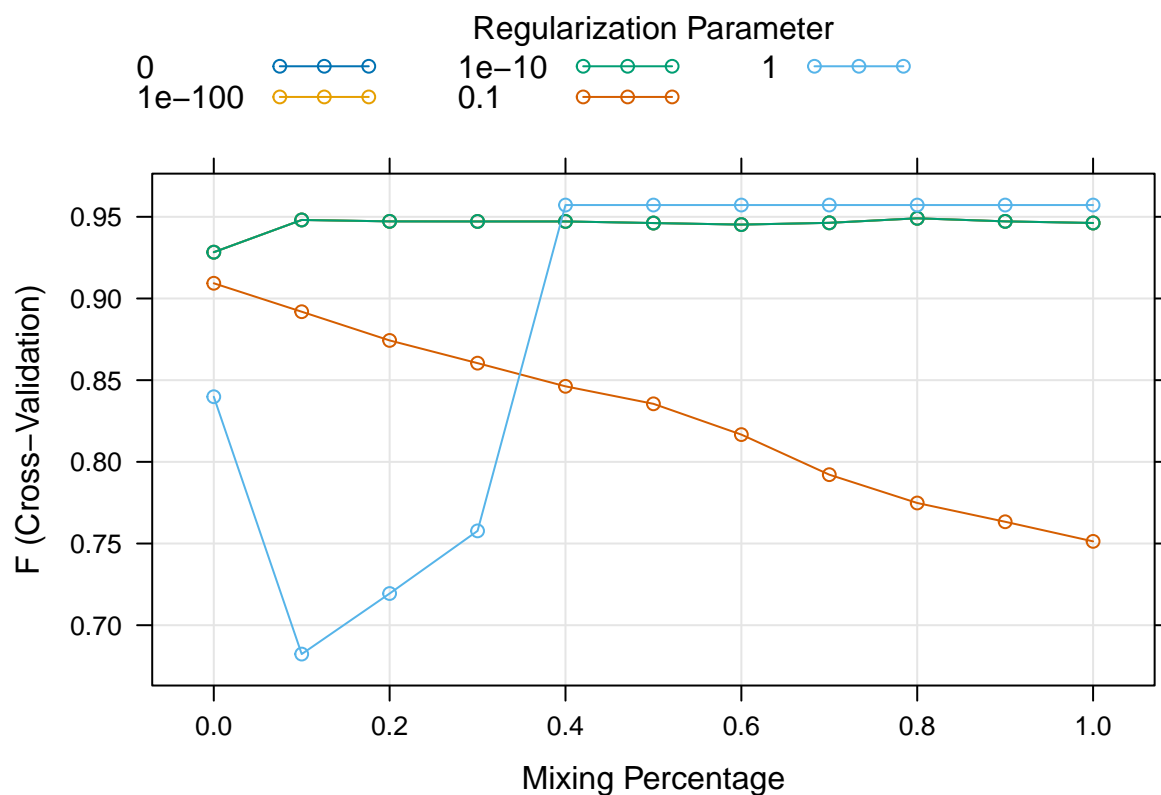
##      alpha lambda prob_threshold      F1
## 5      0.0      1      0.3 0.9687647
## 10     0.1      1      0.4 0.9623554
## 41     0.8      0      0.5 0.9490746
## 11     0.2      0      0.6 0.9480656
## 26     0.5      0      0.7 0.9460062
## 36     0.7      0      0.8 0.9370288

print(extract_best_threshold_f(best_f_3))

##      alpha lambda prob_threshold      F1
## 5      0      1      0.3 0.9687647

plot(logistic_elnet_3)

```



Fitting a logistic elastic net. Grid search of optimal alpha and lambda. Downsampling CV. Maximize F.

```
train_control_4 <- trainControl(
  method = "cv",
  number = 10,
  sampling = "down",
  classProbs = TRUE,
  summaryFunction = prSummary,
  savePredictions = "all",
  seed = cv_seed
)
```

```
logistic_elnet_4 <- train(
  Legendary ~ .,
  data = train_std,
  method = "glmnet",
  family = "binomial",
  metric = "F",
  tuneGrid = grid_,
  trControl = train_control_4,
  intercept = FALSE
)
print(logistic_elnet_4$bestTune)
```

```
##   alpha lambda
## 25   0.4     1
```

```
print(extract_best_f(logistic_elnet_4))
```

```
##      alpha lambda AUC Precision Recall      F AUCSD PrecisionSD RecallSD
## 25    0.4        1  0 0.9178839      1 0.9571637      0 0.008939504      0
##                FSD
## 25 0.004852104
```

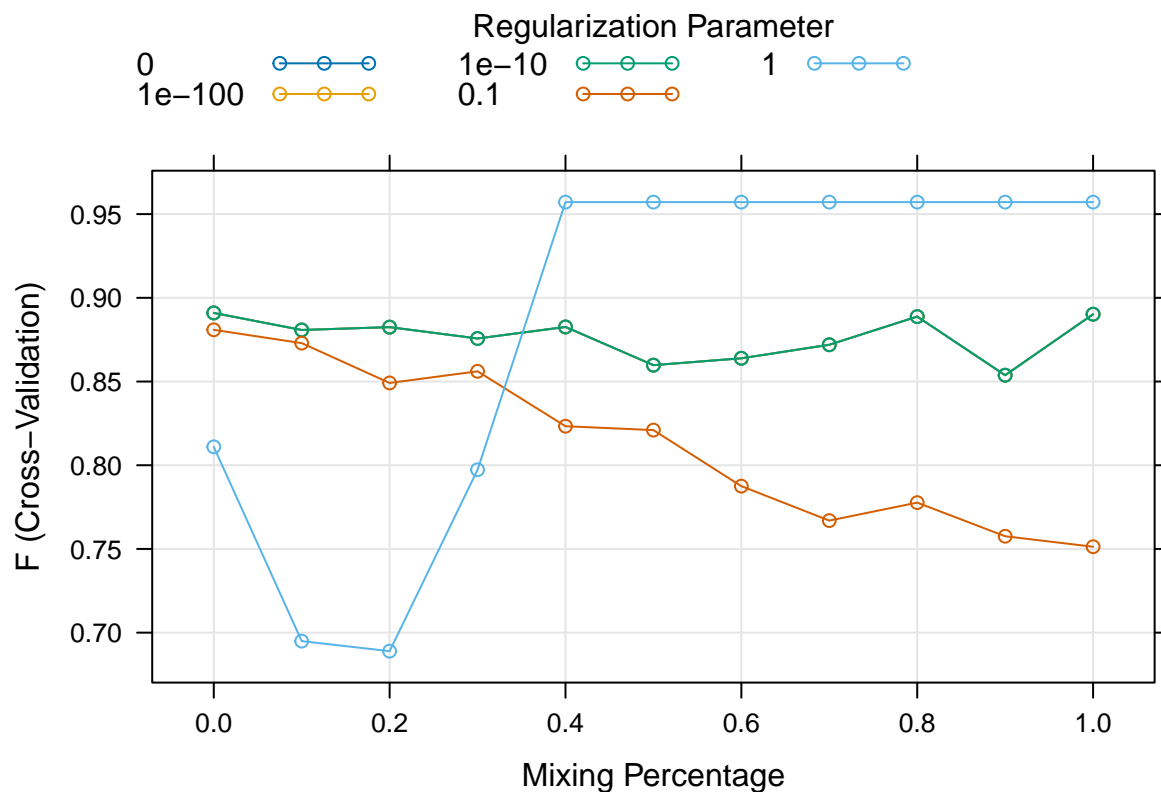
```
best_f_4 <- grid_search_threshold(logistic_elnet_4)
print(best_f_4)
```

```
##      alpha lambda prob_threshold      F1
## 5      0.0        1              0.3 0.9665062
## 10     0.1        1              0.4 0.9650965
## 1      0.0        0              0.5 0.8909693
## 51     1.0        0              0.6 0.8899894
## 511    1.0        0              0.7 0.8851691
## 512    1.0        0              0.8 0.8803295
```

```
print(extract_best_threshold_f(best_f_4))
```

```
##      alpha lambda prob_threshold      F1
## 5      0        1              0.3 0.9665062
```

```
plot(logistic_elnet_4)
```



## Prediction

```
get_prediction <- function(fit, type = "raw") {
  predict(fit, type = type, newdata = test_std |> select(-Legendary))
}

get_accuracy <- function(fit) {
  y <- test_std$Legendary
  y_hat <- predict(fit, type = "raw", newdata = test_std |>
    select(-Legendary))
  return(mean(y == y_hat))
}

lapply(list(logistic_elfnet_1, logistic_elfnet_2, logistic_elfnet_3, logistic_elfnet_4), get_accuracy)

## [[1]]
## [1] 0.9375
##
## [[2]]
## [1] 0.9208333
##
## [[3]]
## [1] 0.9208333
##
## [[4]]
## [1] 0.9208333

get_prediction_thresh <- function(best_f, trainControl) {
  best <- extract_best_threshold_f(best_f)
  alpha <- best$alpha
  lambda <- best$lambda
  tr <- best$prob_threshold
  grid_ <- data.frame(.alpha = alpha, .lambda = lambda)
  fit <- train(
    Legendary ~ .,
    data = train_std,
    method = "glmnet",
    family = "binomial",
    metric = "F",
    tuneGrid = grid_,
    trControl = trainControl,
    intercept = FALSE
  )
  probs <- get_prediction(fit, "prob")[, 2]
  y_hat <- as.factor(ifelse(probs > tr, "Yes", "No"))
  attr(y_hat, "threshold") <- tr
  attr(y_hat, "alpha") <- alpha
  attr(y_hat, "lambda") <- lambda
  return(y_hat)
}

get_accuracy_thresh <- function(y_hat) {
  return(mean(test_std$Legendary == y_hat))
}
```



```
get_prediction_thresh(best_f_1, train_control_1) |> get_accuracy_thresh()
```

```
## [1] 0.925
```

```
get_prediction_thresh(best_f_2, train_control_2) |> get_accuracy_thresh()
```

```
## [1] 0.6875
```

```
get_prediction_thresh(best_f_3, train_control_3) |> get_accuracy_thresh()
```

```
## [1] 0.2458333
```

```
get_prediction_thresh(best_f_4, train_control_4) |> get_accuracy_thresh()
```

```
## [1] 0.2
```

```
get_confusion_matrix <- function(y_hat) {  
  confusionMatrix(y_hat, as.factor(test_std$Legendary))  
}
```

```
get_prediction_thresh(best_f_1, train_control_1) |> get_confusion_matrix()
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction  No Yes
```

```
##           No 219 16
```

```
##           Yes  2  3
```

```
##
```

```
##           Accuracy : 0.925
```

```
##           95% CI : (0.8841, 0.9549)
```

```
## No Information Rate : 0.9208
```

```
## P-Value [Acc > NIR] : 0.465702
```

```
##
```

```
##           Kappa : 0.2244
```

```
##
```

```
## McNemar's Test P-Value : 0.002183
```

```
##
```

```
##           Sensitivity : 0.9910
```

```
##           Specificity : 0.1579
```

```
## Pos Pred Value : 0.9319
```

```
## Neg Pred Value : 0.6000
```

```
## Prevalence : 0.9208
```

```
## Detection Rate : 0.9125
```

```
## Detection Prevalence : 0.9792
```

```
## Balanced Accuracy : 0.5744
```

```
##
```

```
## 'Positive' Class : No
```

```
##
```

```
get_prediction_thresh(best_f_2, train_control_2) |> get_confusion_matrix()
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
```

```
## Prediction  No Yes
```

```
##           No 146  0
```

```
##           Yes  75 19
```

```
##
##          Accuracy : 0.6875
##          95% CI : (0.6247, 0.7456)
##    No Information Rate : 0.9208
##    P-Value [Acc > NIR] : 1
##
##          Kappa : 0.2356
##
##    McNemar's Test P-Value : <2e-16
##
##          Sensitivity : 0.6606
##          Specificity : 1.0000
##    Pos Pred Value : 1.0000
##    Neg Pred Value : 0.2021
##          Prevalence : 0.9208
##    Detection Rate : 0.6083
##    Detection Prevalence : 0.6083
##    Balanced Accuracy : 0.8303
##
##    'Positive' Class : No
##
```

```
get_prediction_thresh(best_f_3, train_control_3) |> get_confusion_matrix()
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction  No Yes
##    No      40  0
##    Yes    181  19
##
##          Accuracy : 0.2458
##          95% CI : (0.1927, 0.3053)
##    No Information Rate : 0.9208
##    P-Value [Acc > NIR] : 1
##
##          Kappa : 0.0338
##
##    McNemar's Test P-Value : <2e-16
##
##          Sensitivity : 0.1810
##          Specificity : 1.0000
##    Pos Pred Value : 1.0000
##    Neg Pred Value : 0.0950
##          Prevalence : 0.9208
##    Detection Rate : 0.1667
##    Detection Prevalence : 0.1667
##    Balanced Accuracy : 0.5905
##
##    'Positive' Class : No
##
```

```
get_prediction_thresh(best_f_4, train_control_4) |> get_confusion_matrix()
```

```
## Confusion Matrix and Statistics
```

```
##
##           Reference
## Prediction  No Yes
##           No   29  0
##           Yes 192 19
##
##           Accuracy : 0.2
##           95% CI : (0.1513, 0.2563)
##           No Information Rate : 0.9208
##           P-Value [Acc > NIR] : 1
##
##           Kappa : 0.0234
##
## Mcnemar's Test P-Value : <2e-16
##
##           Sensitivity : 0.13122
##           Specificity : 1.00000
##           Pos Pred Value : 1.00000
##           Neg Pred Value : 0.09005
##           Prevalence : 0.92083
##           Detection Rate : 0.12083
##           Detection Prevalence : 0.12083
##           Balanced Accuracy : 0.56561
##
##           'Positive' Class : No
##
```

## Extreme Gradient Boosting

### Fitting models

```
library(xgboost)

grid_xgb <- expand.grid(
  nrounds = c(50, 100, 150), # Number of boosting rounds
  eta = c(0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5), # Learning rate
  max_depth = c(3, 6), # Maximum tree depth
  gamma = 0, # Minimum loss reduction; set to default
  colsample_bytree = 1, # Subsample ratio of columns; set to default
  min_child_weight = 1, # Minimum sum of instance weight; set to default
  subsample = 1 # set to default
)

Sys.setenv(DMLC_LOG_FATAL = 1)

xgb_model1 <- train(
  Legendary ~ .,
  data = train_std,
  method = "xgbTree",
  metric = "F",
  tuneGrid = grid_xgb,
  trControl = train_control_1,
  verbose = FALSE
)
```

[illegible]





















[illegible]

```
##      eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 39 0.5          3      0              1              1          1      150
##          AUC Precision      Recall      F          AUCSD PrecisionSD      RecallSD
## 39 0.9626728 0.9628483 0.9843891 0.9731931 0.02166315 0.02785213 0.01544643
##          FSD
## 39 0.01391361
```



[illegible]





















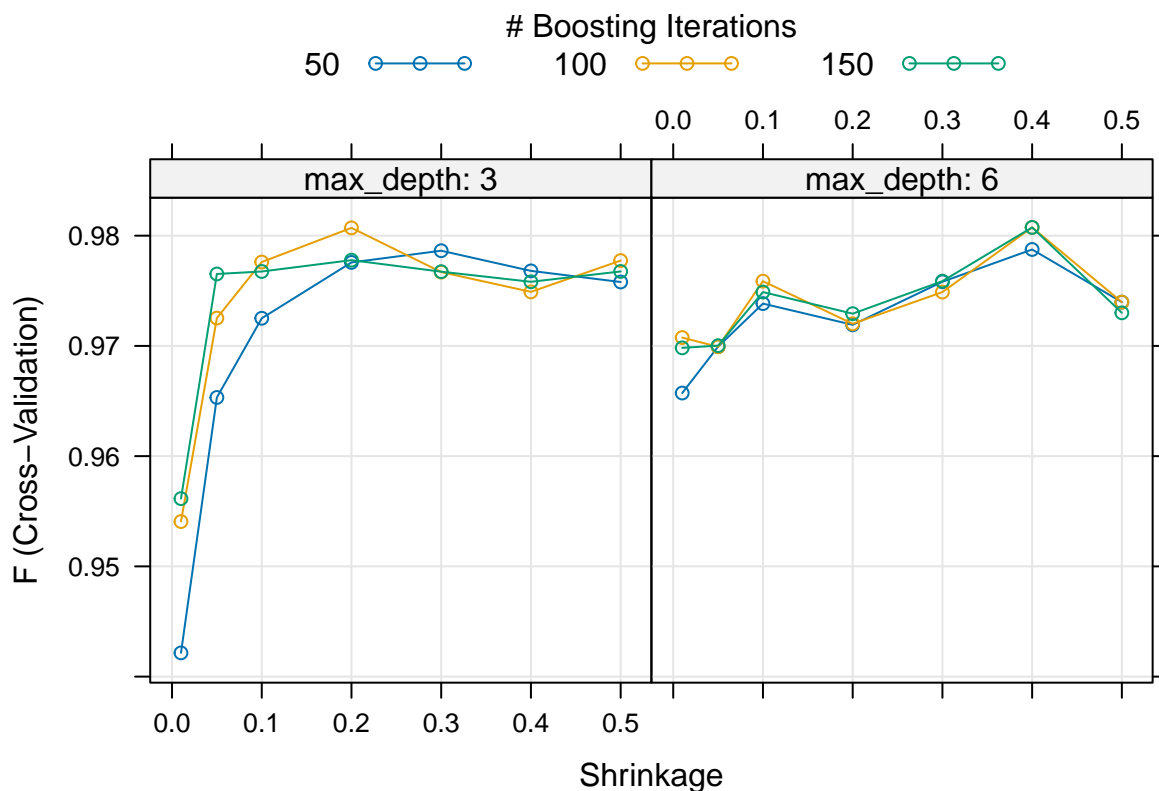


```
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:00] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
```

```
extract_best_f(xgb_model2)
```

```
##      eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 35 0.4          6      0              1              1          1      100
##      AUC Precision      Recall      F      AUCSD PrecisionSD  RecallSD
## 35 0.9396299 0.9735056 0.9883861 0.9807484 0.05261735 0.01804525 0.01349506
##      FSD
## 35 0.01007841
```

```
plot(xgb_model2)
```



## Upsampling

```
Sys.setenv(DMLC_LOG_FATAL = 1)
xgb_model3 <- train(
  Legendary ~ .,
  data = train_std,
  method = "xgbTree",
  metric = "F",
```

)













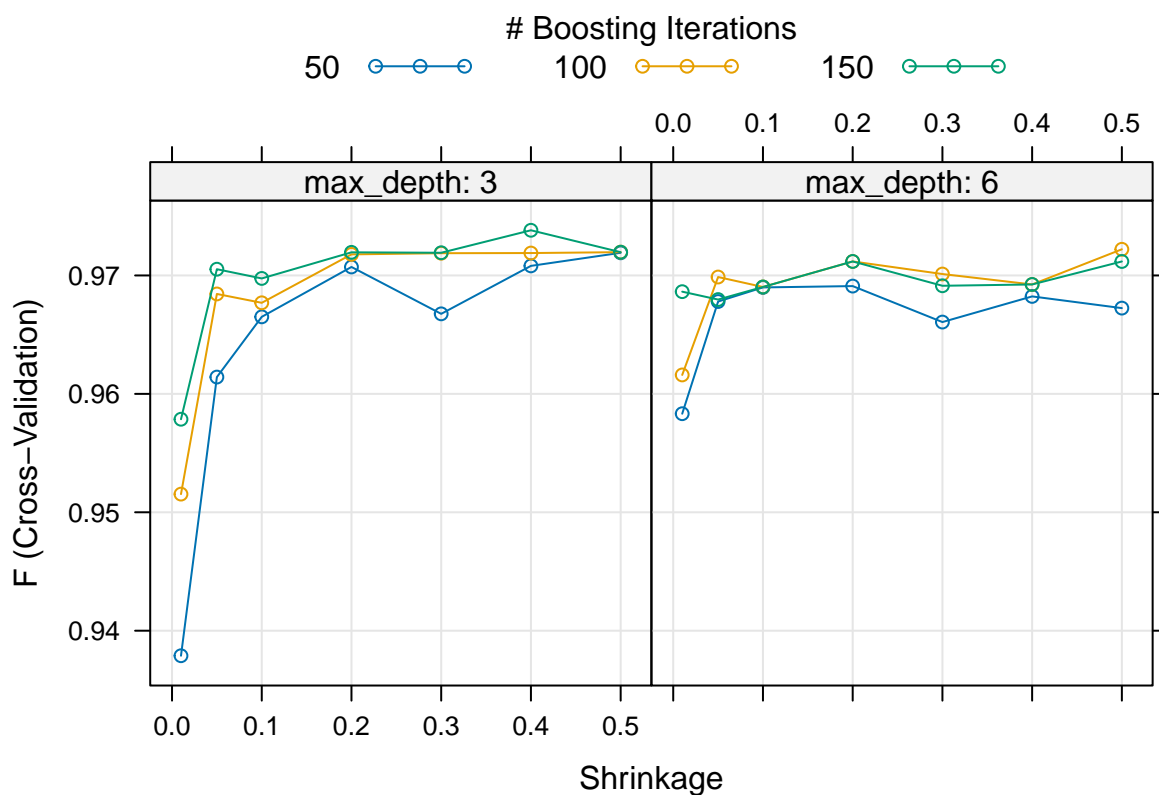












## Downsampling

```
Sys.setenv(DMLC_LOG_FATAL = 1)
xgb_model4 <- train(
  Legendary ~ .,
  data = train_std,
  method = "xgbTree",
  metric = "F",
  tuneGrid = grid_xgb,
  trControl = train_control_4,
  verbose = FALSE
)
```

```
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:30] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
```













[illegible]





[illegible]



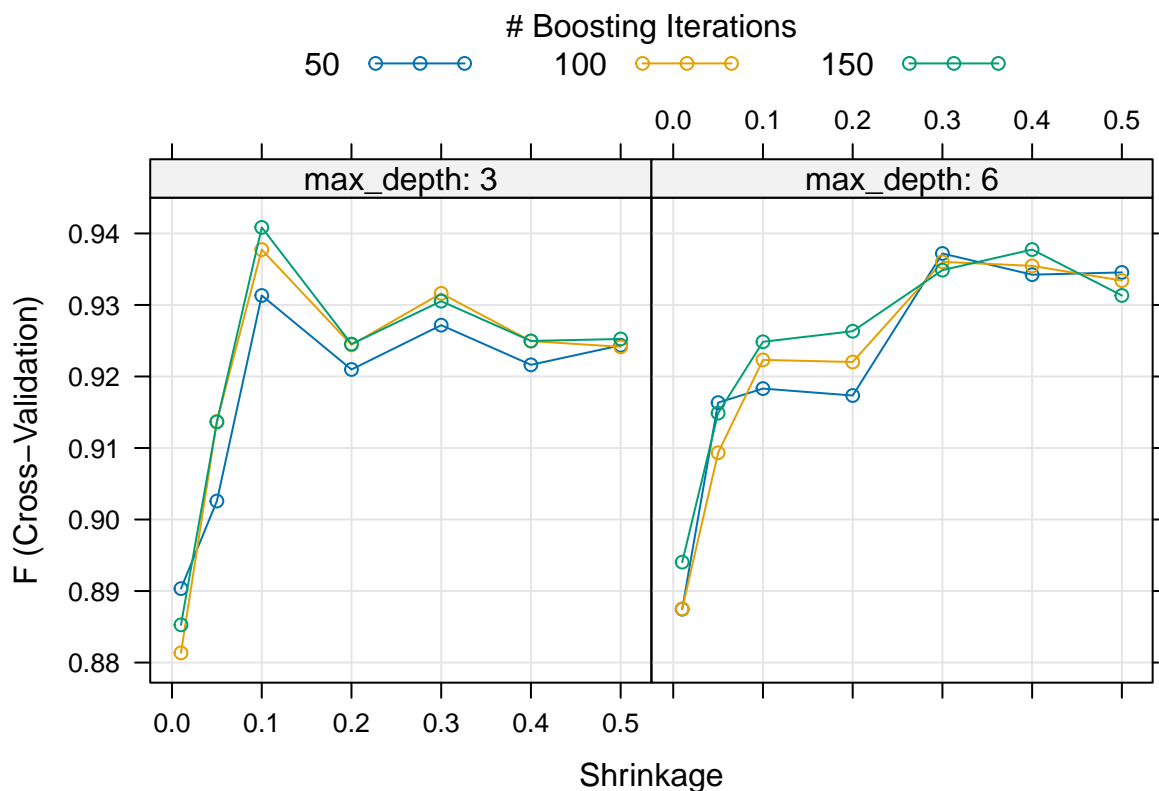


```
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
## [10:05:40] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` instead
```

```
extract_best_f(xgb_model4)
```

```
##      eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 15 0.1          3      0                1                1          1      150
##      AUC Precision   Recall      F      AUCSD PrecisionSD RecallSD
## 15 0.9254964 0.9960784 0.892911 0.940842 0.05269696 0.01240109 0.05072045
##      FSD
## 15 0.02763731
```

```
plot(xgb_model4)
```



## Comparison of F1 Score

```
evaluation_xgb <- as.data.frame(rbind(
  extract_best_f(xgb_model1),
  extract_best_f(xgb_model2),
  extract_best_f(xgb_model3),
  extract_best_f(xgb_model4)
))
```

```

evaluation_xgb <- evaluation_xgb |>
  select(
    -gamma, -colsample_bytree, -min_child_weight, -subsample,
    -AUCSD, -PrecisionSD, -RecallSD, -FSD
  )
rownames(evaluation_xgb) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
kable(evaluation_xgb)

```

	eta	max_depth	nrounds	AUC	Precision	Recall	F
Default-Case	0.5	3	150	0.9626728	0.9628483	0.9843891	0.9731931
SMOTE	0.4	6	100	0.9396299	0.9735056	0.9883861	0.9807484
Up-sampling	0.4	3	150	0.9681308	0.9732789	0.9748115	0.9738142
Down-Sampling	0.1	3	150	0.9254964	0.9960784	0.8929110	0.9408420

- Using SMOTE as sampling method generates the highest F1
- Up-sampling performs similarly to the default case with a good balance of precision and recall, and a strong AUC. It shows that duplicating the minority class can yield comparable performance to the default configuration. The F1 score is slightly worse than for SMOTE
- Down-sampling achieves high precision but at the cost of recall. It gives the lowest F1.

```

# Using different probability thresholds
grid_search_threshold_xgb <- function(fit, thresholds = seq(0.3, 0.8, by = 0.1)) {
  best_given_threshold <- data.frame(matrix(ncol = 9, nrow = 0))

  for (tr in thresholds) {
    # Use the thresholder function to evaluate F1 scores at different thresholds
    res <- thresholder(fit, threshold = tr, final = FALSE, statistics = "F1")
    best <- res[which.max(res$F1), ]
    best_given_threshold <- rbind(best_given_threshold, best)
  }
  best_given_threshold <- best_given_threshold |>
    select(-gamma, -colsample_bytree, -min_child_weight, -subsample)

  return(best_given_threshold)
}

xgb_best_f_1 <- grid_search_threshold_xgb(xgb_model1)
xgb_best_f_2 <- grid_search_threshold_xgb(xgb_model2)
xgb_best_f_3 <- grid_search_threshold_xgb(xgb_model3)
xgb_best_f_4 <- grid_search_threshold_xgb(xgb_model4)

evaluation_xgb <- as.data.frame(rbind(
  extract_best_threshold_f(xgb_best_f_1),
  extract_best_threshold_f(xgb_best_f_2),
  extract_best_threshold_f(xgb_best_f_3),
  extract_best_threshold_f(xgb_best_f_4)
))
rownames(evaluation_xgb) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
kable(evaluation_xgb)

```

	nrounds	max_depth	eta	prob_threshold	F1
Default-Case	150	3	0.30	0.3	0.9762566
SMOTE	100	6	0.40	0.5	0.9807484
Up-sampling	50	3	0.40	0.4	0.9748808
Down-Sampling	50	3	0.01	0.3	0.9571795

- The model performs well without special sampling. The F1 score is high, indicating a good balance between precision and recall.
- Using SMOTE improves the F1 score. This suggests that balancing the classes helps the model better detect rare classes i.e. the sampling method works well when the dataset is imbalanced.
- Up-sampling also performs well but slightly worse than SMOTE. It indicates that repeating the minority class data helps the model.
- Down-sampling yields the lowest F1 score, implying that reducing the number of majority class samples to balance the dataset may lead to some loss of information and a less effective model.

## Prediction

```
xgb_models <- list(xgb_model1, xgb_model2, xgb_model3, xgb_model4)
accuracy_xgb <- do.call(rbind, lapply(xgb_models, get_accuracy))
rownames(accuracy_xgb) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
colnames(accuracy_xgb) <- "Accuracy"
kable(accuracy_xgb)
```

	Accuracy
Default-Case	0.9375000
SMOTE	0.9333333
Up-sampling	0.9250000
Down-Sampling	0.8958333

- The model without any special sampling technique achieved the highest accuracy.
- Up-sampling gives the second highest accuracy. It seems like duplicating the minority class (Legendary="yes") helped improved model performance slightly over SMOTE.
- The model using down-sampling gives the lowest accuracy - reducing the size of the majority class (Legendary="No") might have caused a loss in important information.

## Testing different prob. thresholds

```
get_prediction_thresh_xgb <- function(best_f, trainControl) {
  # extract the best threshold from the results and the parameters
  best <- extract_best_threshold_f(best_f)
  eta <- best$eta
  max_depth <- best$max_depth
  nrounds <- best$nrounds
  tr <- best$prob_threshold

  # define the tuning grid
  grid_xgb <- expand.grid(
    nrounds = nrounds,
    eta = eta,
    max_depth = max_depth,
```

```

    gamma = 0, # Default value
    colsample_bytree = 1, # Default value
    min_child_weight = 1, # Default value
    subsample = 1 # Default value
  )

  # train the model
  fit_xgb <- train(
    Legendary ~ .,
    data = train_std,
    method = "xgbTree",
    metric = "F",
    tuneGrid = grid_xgb,
    trControl = trainControl,
    verbose = FALSE
  )

  # get the predicted probabilities
  probs <- predict(fit_xgb, newdata = test_std, type = "prob")[, 2]

  # apply the threshold
  y_hat <- as.factor(ifelse(probs > tr, "Yes", "No"))

  # set levels of y_hat to match test_std$Legendary
  y_hat <- factor(y_hat, levels = levels(test_std$Legendary))

  # store as attributes of y_hat
  attr(y_hat, "threshold") <- tr
  attr(y_hat, "eta") <- eta
  attr(y_hat, "max_depth") <- max_depth
  attr(y_hat, "nrounds") <- nrounds

  return(y_hat)
}

acc_1 <- get_prediction_thresh_xgb(xgb_best_f_1, train_control_1) |> get_accuracy_thresh()
acc_2 <- get_prediction_thresh_xgb(xgb_best_f_2, train_control_2) |> get_accuracy_thresh()
acc_3 <- get_prediction_thresh_xgb(xgb_best_f_3, train_control_3) |> get_accuracy_thresh()
acc_4 <- get_prediction_thresh_xgb(xgb_best_f_4, train_control_4) |> get_accuracy_thresh()

accuracy_xgb <- rbind(acc_1, acc_2, acc_3, acc_4)
rownames(accuracy_xgb) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
colnames(accuracy_xgb) <- "Accuracy"
kable(accuracy_xgb)

```

	Accuracy
Default-Case	0.9375000
SMOTE	0.9333333
Up-sampling	0.9208333
Down-Sampling	0.0791667

The significant drop in accuracy for the model using down-sampling is caused by the lower threshold used to

maximize the F1-score. The lower threshold leads to all predictions being equal to “Yes” (=minority class) i.e. the lower applies threshold leads to an increasing bias toward predicting “Yes”.

```
y_hat_xgb_4 <- get_prediction_thresh_xgb(xgb_best_f_4, train_control_4)
confusionMatrix(y_hat_xgb_4, test_std$Legendary)
```

```
## Confusion Matrix and Statistics
##
##              Reference
## Prediction  No  Yes
##          No    0   0
##          Yes 221  19
##
##              Accuracy : 0.0792
##              95% CI : (0.0483, 0.1209)
##          No Information Rate : 0.9208
##          P-Value [Acc > NIR] : 1
##
##              Kappa : 0
##
## Mcnemar's Test P-Value : <2e-16
##
##              Sensitivity : 0.00000
##              Specificity : 1.00000
##          Pos Pred Value :      NaN
##          Neg Pred Value : 0.07917
##              Prevalence : 0.92083
##          Detection Rate : 0.00000
##          Detection Prevalence : 0.00000
##          Balanced Accuracy : 0.50000
##
##          'Positive' Class : No
##
```

## Random Forest

```
library(randomForest)
## parallelize the computational process
library(doParallel)
cl <- makePSOCKcluster(5)
registerDoParallel(cl)
```

Fitting a random forest. Grid search of optimal mtry. CV. Maximize F.

```
grid_rf <- expand.grid(.mtry = 1:28) # mtry -- number of variables randomly sampled as candidates at ea
rf_model1 <- train(Legendary ~ .,
  data = train_std,
  method = "rf",
  metric = "F",
  tuneGrid = grid_rf,
  trcontrol = train_control_1,
  verbose = FALSE,
  proximity = FALSE,
```

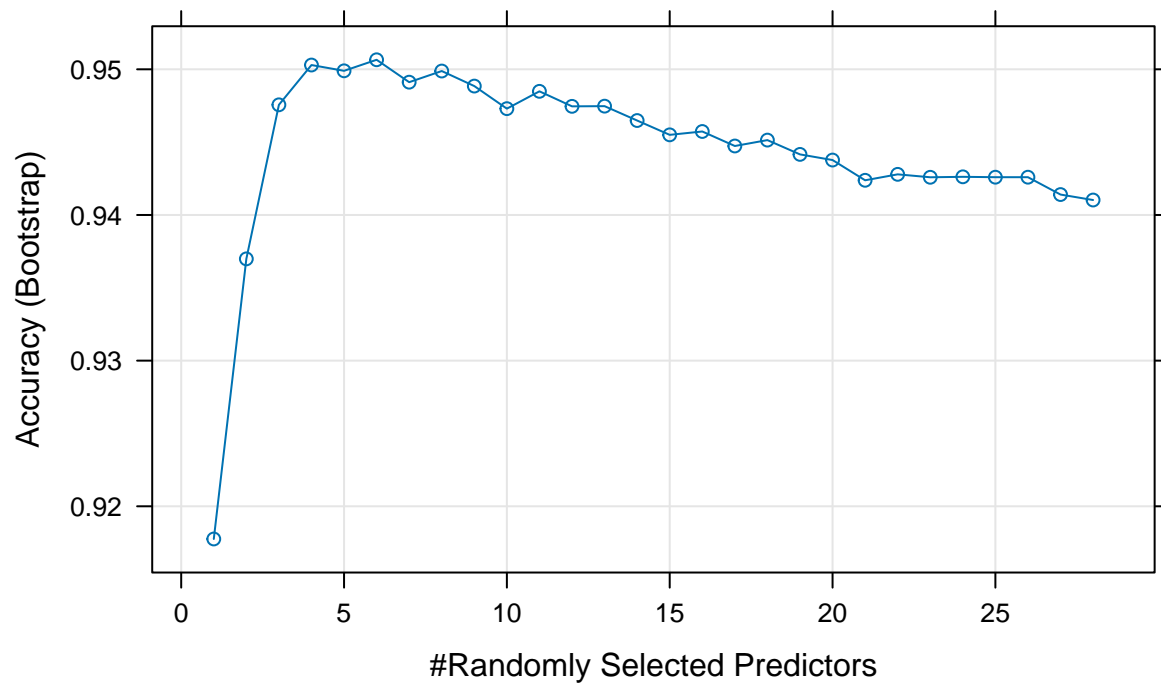
```

    importance = TRUE
)
# summary of the model
print(rf_model1)

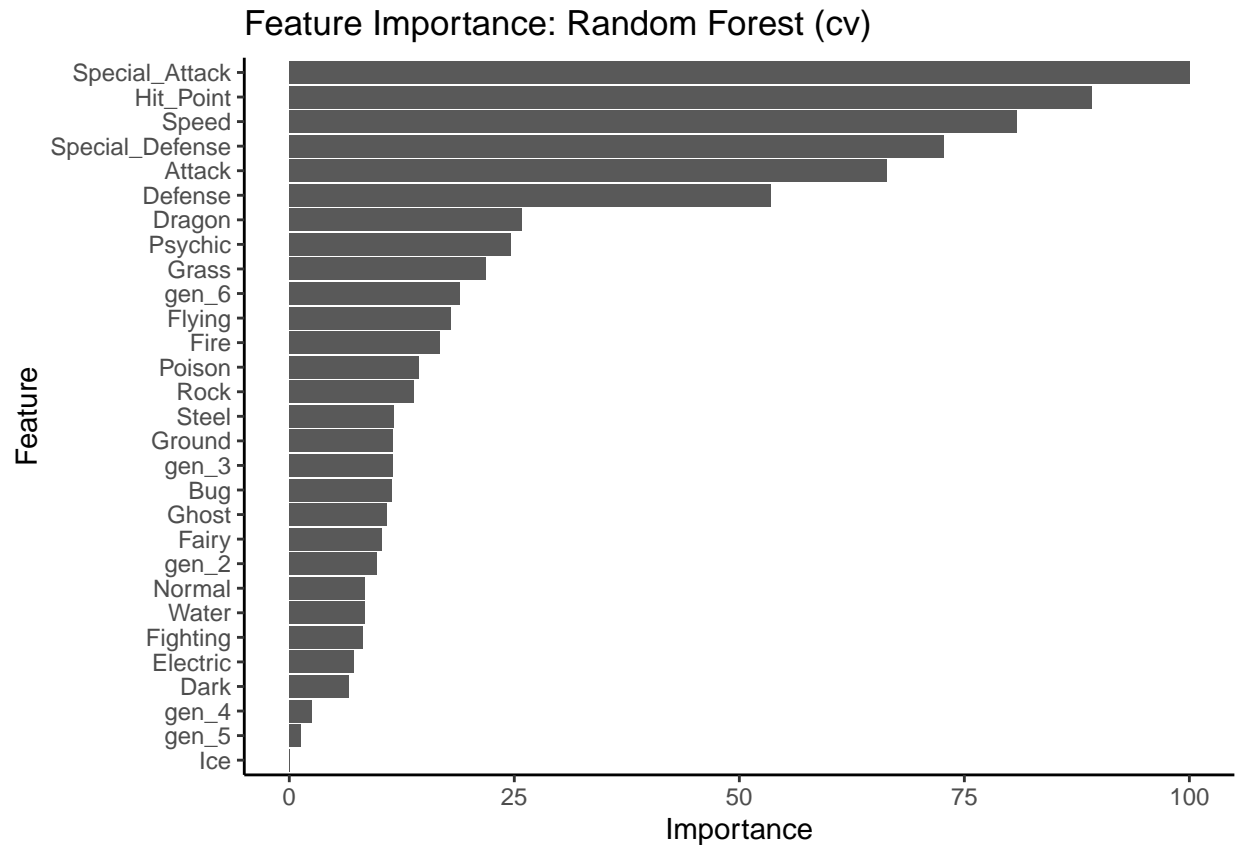
## Random Forest
##
## 560 samples
## 29 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 560, 560, 560, 560, 560, 560, ...
## Resampling results across tuning parameters:
##
##  mtry  Accuracy  Kappa
##  1     0.9177565  0.0000000
##  2     0.9369860  0.3768630
##  3     0.9475623  0.5291413
##  4     0.9502938  0.5670435
##  5     0.9498966  0.5692307
##  6     0.9506545  0.5786305
##  7     0.9491157  0.5679341
##  8     0.9498811  0.5828189
##  9     0.9488496  0.5803627
## 10     0.9472987  0.5678013
## 11     0.9484879  0.5847877
## 12     0.9474584  0.5776187
## 13     0.9474700  0.5809676
## 14     0.9464842  0.5728921
## 15     0.9455003  0.5735916
## 16     0.9457322  0.5738526
## 17     0.9447354  0.5696412
## 18     0.9451403  0.5741988
## 19     0.9441591  0.5687198
## 20     0.9437704  0.5640706
## 21     0.9423809  0.5565452
## 22     0.9427891  0.5632986
## 23     0.9425885  0.5574343
## 24     0.9426195  0.5640811
## 25     0.9425932  0.5687365
## 26     0.9425933  0.5684606
## 27     0.9414016  0.5617561
## 28     0.9410264  0.5609319
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 6.
plot(rf_model1, main = "\n # Variable selection: Random Forest")

```

## # Variable selection: Random Forest



```
# feature importance
ggplot(varImp(rf_model1), aes(
  x = reorder(feature, Importance),
  y = Importance
)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  theme_classic() +
  labs(
    x = "Feature",
    y = "Importance",
    title = "Feature Importance: Random Forest (cv)"
  )
)
```



```
## manually find the best ntree parameter
tuneGrid1 <- expand.grid(.mtry = 26)
modellist1 <- list()
```

```
# train with different ntree parameters
for (ntree in c(500, 1500, 2500, 5000)) {
  fit <- train(Legendary ~ .,
    data = train_std,
    method = "rf",
    metric = "F",
    tuneGrid = tuneGrid1,
    trControl = train_control_1,
    ntree = ntree
  )
  key <- toString(ntree)
  modellist1[[key]] <- fit
}
```

```
# Compare results
results1 <- resamples(modellist1)
summary(results1)
```

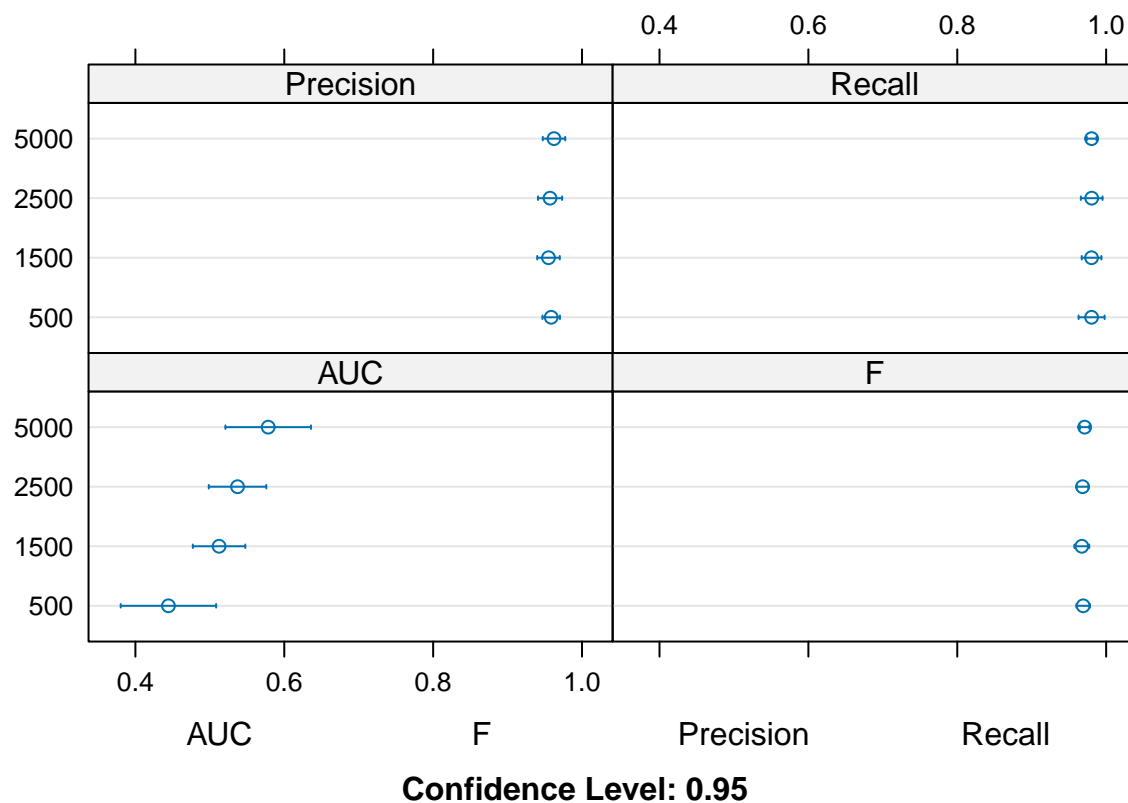
```
##
## Call:
## summary.resamples(object = results1)
##
```



```

## Models: 500, 1500, 2500, 5000
## Number of resamples: 10
##
## AUC
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.3215042 0.3651514 0.4374092 0.4443782 0.5177106 0.5686275    0
## 1500 0.4472397 0.4814497 0.4990622 0.5124026 0.5469615 0.6051212    0
## 2500 0.4608195 0.4868232 0.5462813 0.5372172 0.5743788 0.6059073    0
## 5000 0.4289684 0.5518272 0.5788910 0.5784059 0.6415895 0.6730769    0
##
## F
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9433962 0.9603846 0.9714182 0.9691574 0.9807692 0.9811321    0
## 1500 0.9411765 0.9618946 0.9705854 0.9673311 0.9716904 0.9904762    0
## 2500 0.9514563 0.9617199 0.9714286 0.9683621 0.9779907 0.9807692    0
## 5000 0.9523810 0.9708738 0.9711512 0.9711834 0.9782848 0.9803922    0
##
## Precision
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9259259 0.9494755 0.9622642 0.9585953 0.9629630 0.9795918    0
## 1500 0.9259259 0.9417314 0.9534965 0.9549503 0.9753846 0.9811321    0
## 2500 0.9259259 0.9446970 0.9611614 0.9570341 0.9622642 1.0000000    0
## 5000 0.9259259 0.9494755 0.9615385 0.9623489 0.9758602 1.0000000    0
##
## Recall
##           Min.    1st Qu.    Median    Mean    3rd Qu. Max. NA's
## 500  0.9411765 0.9662519 0.9901961 0.9804676 1.0000000    1    0
## 1500 0.9411765 0.9803922 0.9803922 0.9804299 0.9951923    1    0
## 2500 0.9423077 0.9662519 0.9805807 0.9806184 1.0000000    1    0
## 5000 0.9615385 0.9803922 0.9803922 0.9805430 0.9806750    1    0
dotplot(results1) # highest Precision-Recall score when ntrees=2500, mtry=26

```



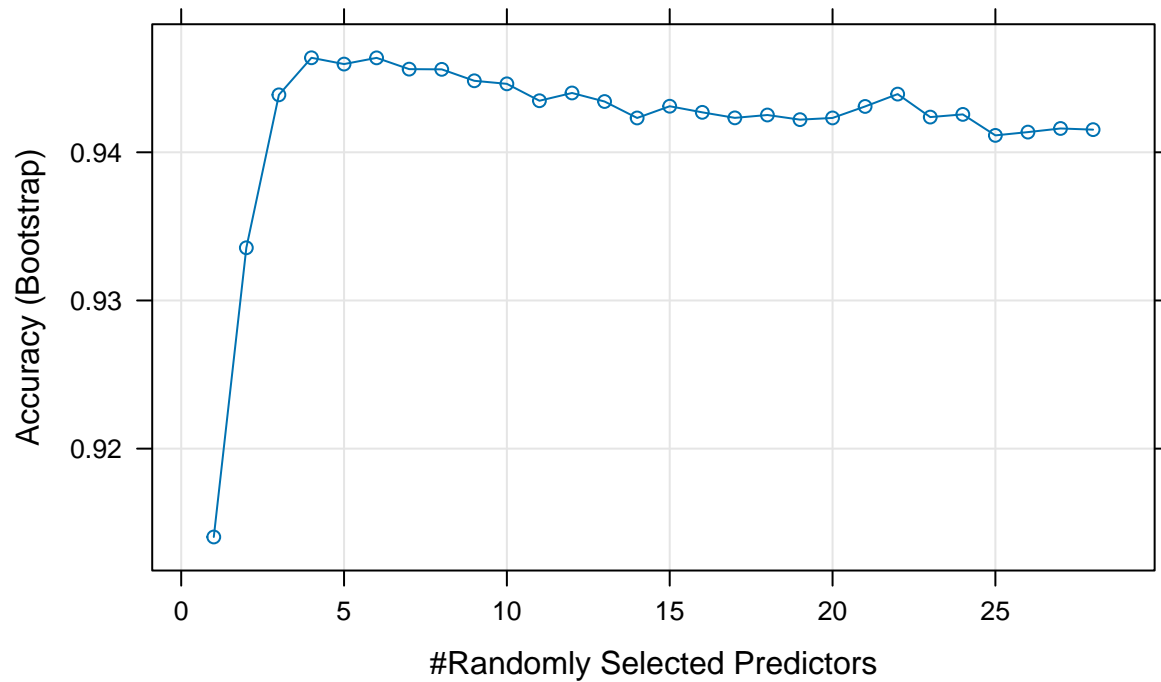
Fitting a random forest. Grid search of optimal mtry. SMOTE CV. Maximize F.

```
rf_model2 <- train(Legendary ~ .,
  data = train_std,
  method = "rf",
  metric = "F",
  tuneGrid = grid_rf,
  trcontrol = train_control_2,
  verbose = FALSE,
  proximity = FALSE,
  importance = TRUE
)
# summary of the model
print(rf_model2)

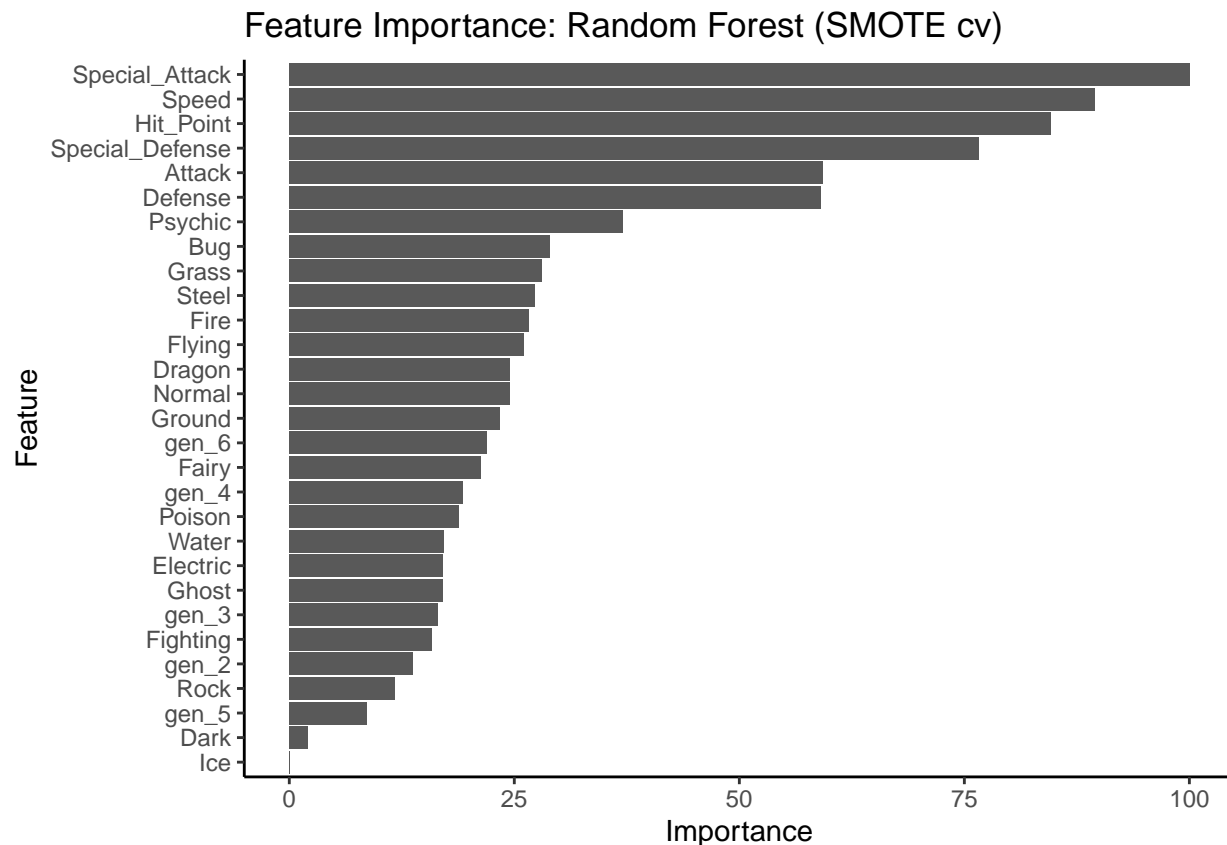
## Random Forest
##
## 560 samples
## 29 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 560, 560, 560, 560, 560, ...
## Resampling results across tuning parameters:
##
```

```
##      mtry Accuracy  Kappa
##      1  0.9140385 0.0000000
##      2  0.9335551 0.3773052
##      3  0.9438773 0.5211415
##      4  0.9463787 0.5597343
##      5  0.9459537 0.5626954
##      6  0.9463738 0.5731539
##      7  0.9456112 0.5693021
##      8  0.9455966 0.5724802
##      9  0.9448247 0.5690735
##     10  0.9446210 0.5677984
##     11  0.9434795 0.5659047
##     12  0.9440020 0.5707156
##     13  0.9434265 0.5670937
##     14  0.9423163 0.5589432
##     15  0.9431086 0.5685024
##     16  0.9426956 0.5680620
##     17  0.9423244 0.5627667
##     18  0.9425162 0.5675255
##     19  0.9422023 0.5692595
##     20  0.9423191 0.5730225
##     21  0.9430937 0.5738291
##     22  0.9439268 0.5821477
##     23  0.9423765 0.5751364
##     24  0.9425583 0.5783075
##     25  0.9411407 0.5660532
##     26  0.9413629 0.5712101
##     27  0.9416073 0.5735300
##     28  0.9415291 0.5692022
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 4.
plot(rf_model2, main = "\n # Variable selection: Random Forest")
```

## # Variable selection: Random Forest



```
# feature importance
ggplot(varImp(rf_model2), aes(
  x = reorder(feature, Importance),
  y = Importance
)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  theme_classic() +
  labs(
    x = "Feature",
    y = "Importance",
    title = "Feature Importance: Random Forest (SMOTE cv)"
  )
)
```



```
## manually find the best ntree parameter
tuneGrid2 <- expand.grid(.mtry = 8)
modellist2 <- list()
```

```
# train with different ntree parameters
for (ntree in c(500, 1500, 2500, 5000)) {
  fit <- train(Legendary ~ .,
    data = train_std,
    method = "rf",
    metric = "F",
    tuneGrid = tuneGrid2,
    trControl = train_control_2,
    ntree = ntree
  )
  key <- toString(ntree)
  modellist2[[key]] <- fit
}
```

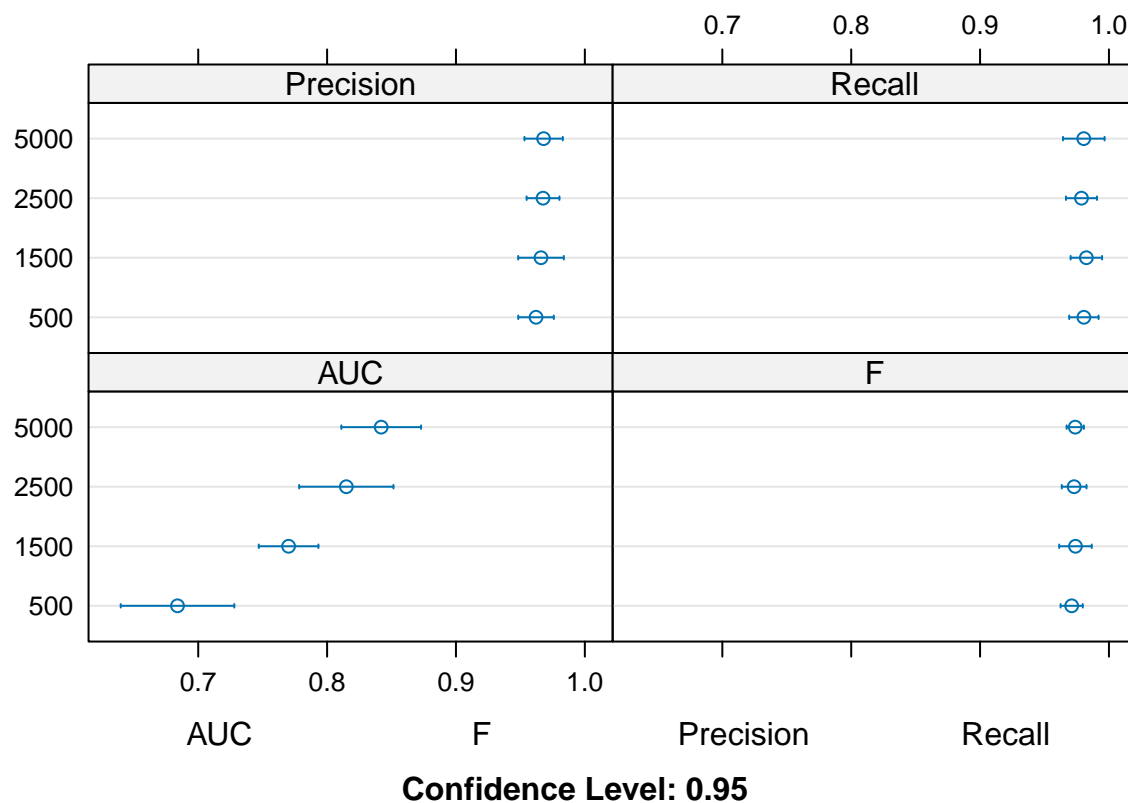
```
# Compare results
results2 <- resamples(modellist2)
summary(results2)
```

```
##
## Call:
## summary.resamples(object = results2)
##
```

```

## Models: 500, 1500, 2500, 5000
## Number of resamples: 10
##
## AUC
##      Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.5490196 0.6684088 0.6954026 0.6839108 0.7094545 0.7623968    0
## 1500 0.7247432 0.7446686 0.7734848 0.7701167 0.7862687 0.8235294    0
## 2500 0.7101063 0.7877638 0.8241124 0.8149155 0.8543479 0.8770728    0
## 5000 0.7624175 0.8406088 0.8435862 0.8419895 0.8569043 0.9023729    0
##
## F
##      Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9523810 0.9615385 0.9708628 0.9710930 0.9807692 0.9902913    0
## 1500 0.9423077 0.9708738 0.9716956 0.9740247 0.9877666 1.0000000    0
## 2500 0.9514563 0.9708738 0.9711512 0.9729816 0.9807692 0.9902913    0
## 5000 0.9600000 0.9702970 0.9714182 0.9738464 0.9806750 0.9904762    0
##
## Precision
##      Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9259259 0.9485294 0.9622507 0.9621360 0.9805769 0.9807692    0
## 1500 0.9245283 0.9494755 0.9622642 0.9659882 0.9809471 1.0000000    0
## 2500 0.9433962 0.9609729 0.9615385 0.9675548 0.9807692 1.0000000    0
## 5000 0.9272727 0.9617199 0.9712774 0.9680217 0.9800000 1.0000000    0
##
## Recall
##      Min.    1st Qu.    Median    Mean    3rd Qu. Max. NA's
## 500  0.9607843 0.9662519 0.9805807 0.9805430 0.9951923    1    0
## 1500 0.9607843 0.9662519 0.9805807 0.9824661 1.0000000    1    0
## 2500 0.9423077 0.9803922 0.9803922 0.9786953 0.9807692    1    0
## 5000 0.9411765 0.9609729 0.9901961 0.9804676 1.0000000    1    0
dotplot(results2) # highest Precision-Recall score when ntree=500, mtry=8

```



Fitting a random forest. Grid search of optimal mtry. Upsampling CV. Maximize F.

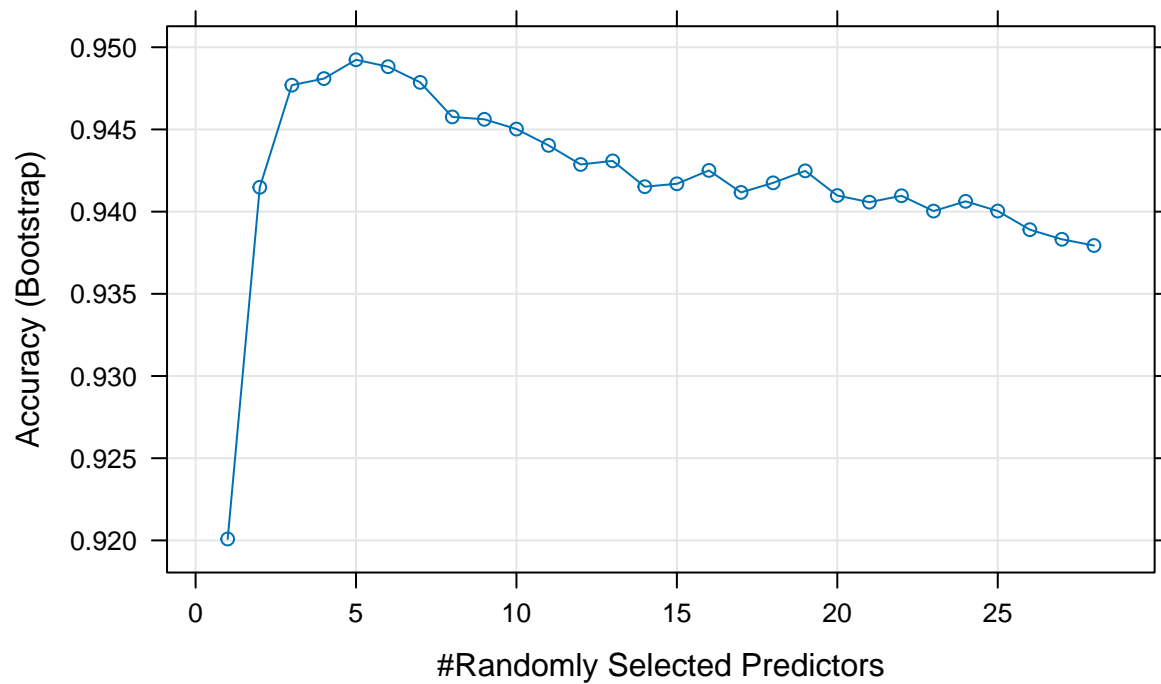
```
rf_model3 <- train(Legendary ~ .,
  data = train_std,
  method = "rf",
  metric = "F",
  tuneGrid = grid_rf,
  trcontrol = train_control_3,
  verbose = FALSE,
  proximity = FALSE,
  importance = TRUE
)
# summary of the model
print(rf_model3)
```

```
## Random Forest
##
## 560 samples
## 29 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 560, 560, 560, 560, 560, 560, ...
```

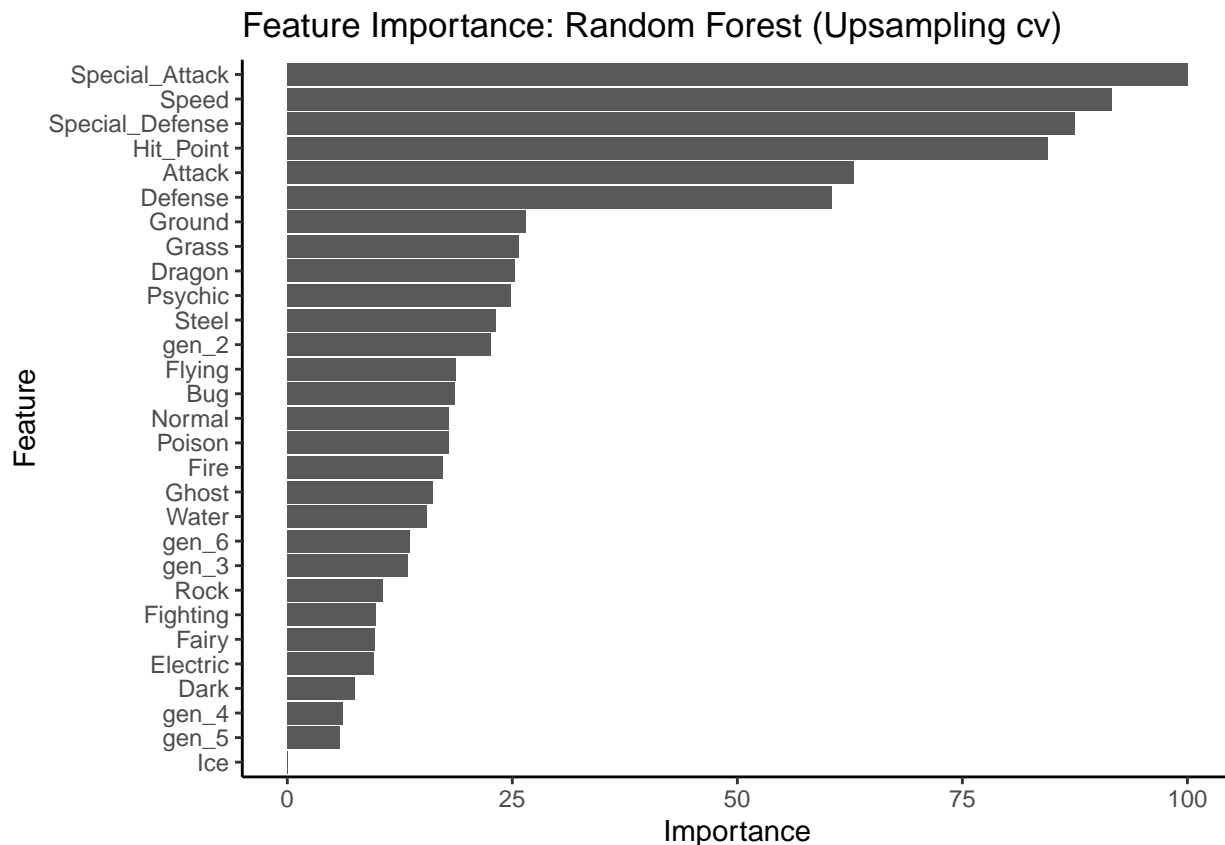
```
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##    1    0.9200872  0.0000000
##    2    0.9414825  0.4128568
##    3    0.9476939  0.5218870
##    4    0.9480941  0.5339410
##    5    0.9492380  0.5538915
##    6    0.9488176  0.5551127
##    7    0.9478631  0.5508851
##    8    0.9457617  0.5442549
##    9    0.9456155  0.5460851
##   10    0.9450235  0.5468262
##   11    0.9440313  0.5401639
##   12    0.9428697  0.5391706
##   13    0.9430887  0.5394927
##   14    0.9415183  0.5342469
##   15    0.9416942  0.5322142
##   16    0.9425078  0.5378427
##   17    0.9411693  0.5329109
##   18    0.9417506  0.5325291
##   19    0.9424819  0.5402458
##   20    0.9409787  0.5312908
##   21    0.9405749  0.5312745
##   22    0.9409655  0.5366852
##   23    0.9400351  0.5323622
##   24    0.9406273  0.5388440
##   25    0.9400382  0.5324809
##   26    0.9389022  0.5276088
##   27    0.9383177  0.5262461
##   28    0.9379391  0.5280452
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 5.
plot(rf_model3, main = "\n # Variable selection: Random Forest")
```



## # Variable selection: Random Forest



```
# feature importance
ggplot(varImp(rf_model3), aes(
  x = reorder(feature, Importance),
  y = Importance
)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  theme_classic() +
  labs(
    x = "Feature",
    y = "Importance",
    title = "Feature Importance: Random Forest (Upsampling cv)"
  )
)
```



```
## manually find the best ntree parameter
tuneGrid3 <- expand.grid(.mtry = 25)
modellist3 <- list()
```

```
# train with different ntree parameters
for (ntree in c(500, 1500, 2500, 5000)) {
  fit <- train(Legendary ~ .,
    data = train_std,
    method = "rf",
    metric = "F",
    tuneGrid = tuneGrid3,
    trControl = train_control_3,
    ntree = ntree
  )
  key <- toString(ntree)
  modellist3[[key]] <- fit
}
```

```
# Compare results
results3 <- resamples(modellist3)
summary(results3)
```

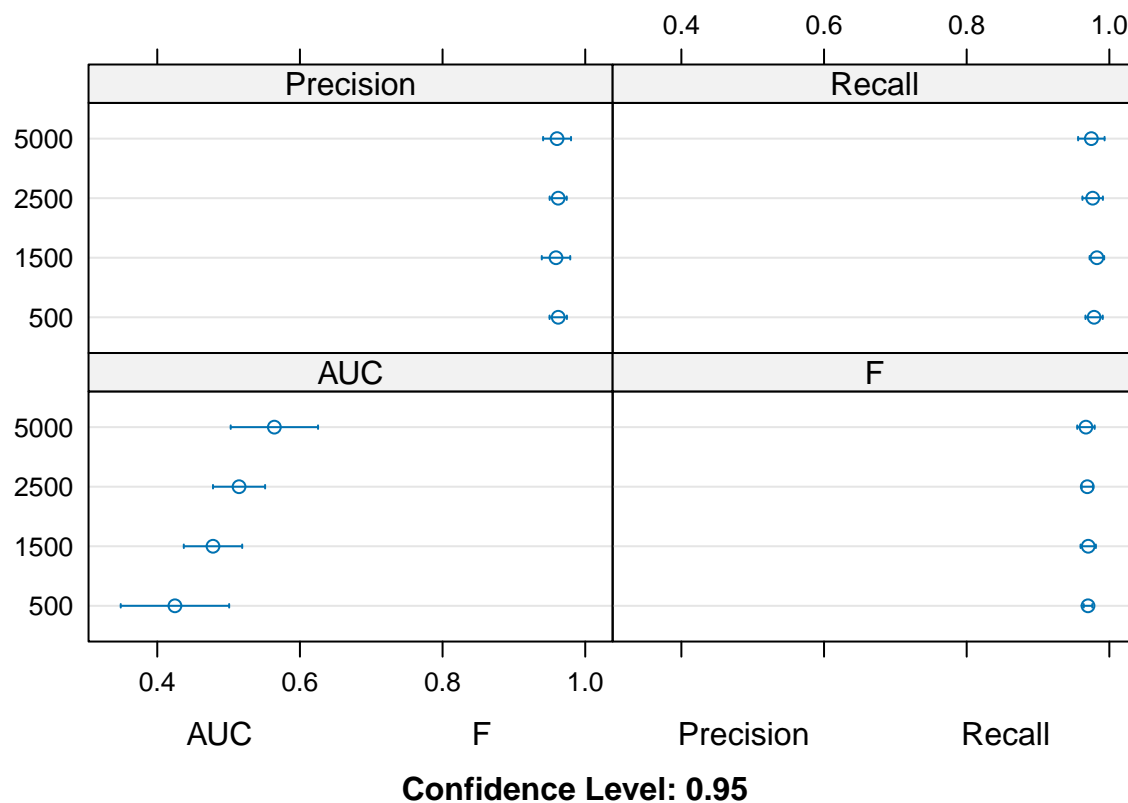
```
##
## Call:
## summary.resamples(object = results3)
##
```

```

## Models: 500, 1500, 2500, 5000
## Number of resamples: 10
##
## AUC
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.2972305  0.3362789  0.4100528  0.4247800  0.4874577  0.6299310    0
## 1500 0.4035517  0.4323867  0.4700243  0.4781435  0.5217662  0.5737761    0
## 2500 0.4306255  0.4791879  0.5071932  0.5146695  0.5618853  0.5817184    0
## 5000 0.4846546  0.5084443  0.5236422  0.5641802  0.5849494  0.7243549    0
##
## F
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9523810  0.9704412  0.9711512  0.9701020  0.9714286  0.9811321    0
## 1500 0.9523810  0.9555193  0.9708738  0.9703781  0.9806750  0.9904762    0
## 2500 0.9514563  0.9647965  0.9705854  0.9690888  0.9782954  0.9811321    0
## 5000 0.9423077  0.9549533  0.9661064  0.9671838  0.9781513  1.0000000    0
##
## Precision
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9259259  0.9615385  0.9622642  0.9621331  0.9757407  0.9803922    0
## 1500 0.9107143  0.9436583  0.9619013  0.9590200  0.9803922  1.0000000    0
## 2500 0.9285714  0.9609729  0.9619013  0.9620737  0.9757407  0.9807692    0
## 5000 0.9107143  0.9485294  0.9622642  0.9604984  0.9798980  1.0000000    0
##
## Recall
##           Min.    1st Qu.    Median    Mean    3rd Qu. Max. NA's
## 500  0.9423077  0.9803922  0.9803922  0.9786199  0.9807692    1    0
## 1500 0.9615385  0.9803922  0.9803922  0.9825415  0.9951923    1    0
## 2500 0.9423077  0.9607843  0.9803922  0.9766214  0.9951923    1    0
## 5000 0.9411765  0.9469268  0.9805807  0.9747738  1.0000000    1    0

```

`dotplot(results3)` # *highest Precision-Recall score when ntree=500, mtry=25*



Fitting a random forest. Grid search of optimal mtry. Downsampling CV. Maximize F.

```
rf_model4 <- train(Legendary ~ .,
  data = train_std,
  method = "rf",
  metric = "F",
  tuneGrid = grid_rf,
  trcontrol = train_control_4,
  verbose = FALSE,
  proximity = FALSE,
  importance = TRUE
)
# summary of the model
print(rf_model4)
```

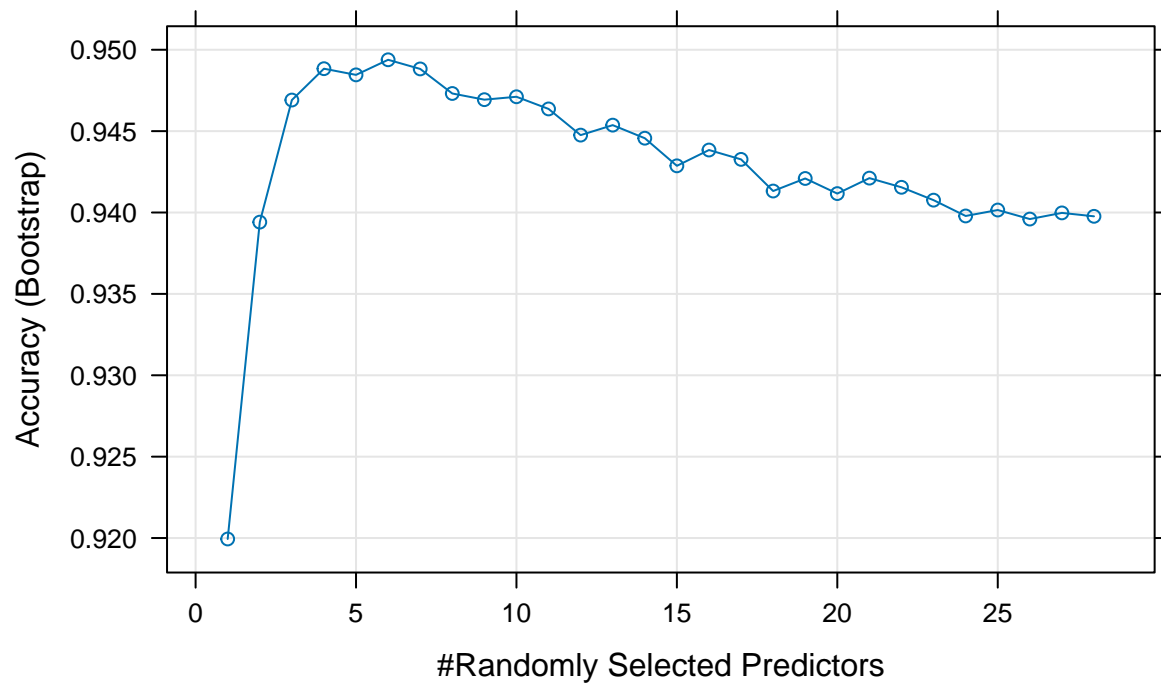
```
## Random Forest
##
## 560 samples
## 29 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 560, 560, 560, 560, 560, 560, ...
```

```

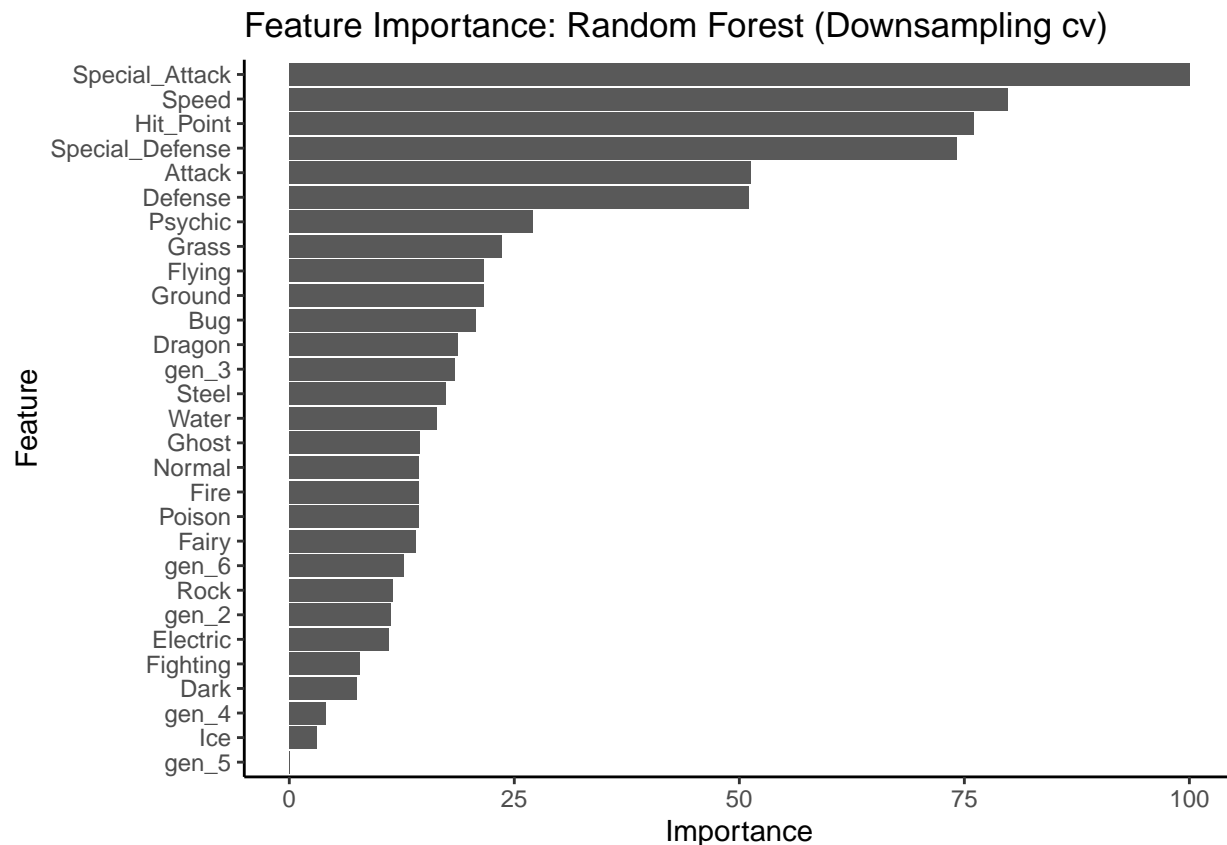
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##    1    0.9199421  0.0000000
##    2    0.9394099  0.3808127
##    3    0.9469103  0.4979858
##    4    0.9488368  0.5271101
##    5    0.9484562  0.5350315
##    6    0.9493840  0.5434702
##    7    0.9488223  0.5446154
##    8    0.9473121  0.5393034
##    9    0.9469329  0.5362799
##   10    0.9471109  0.5452008
##   11    0.9463647  0.5478410
##   12    0.9447553  0.5349908
##   13    0.9453682  0.5420684
##   14    0.9445650  0.5443173
##   15    0.9428679  0.5265960
##   16    0.9438394  0.5434632
##   17    0.9432642  0.5421173
##   18    0.9413199  0.5258892
##   19    0.9420920  0.5369727
##   20    0.9411626  0.5278834
##   21    0.9421118  0.5349435
##   22    0.9415484  0.5339920
##   23    0.9407585  0.5288871
##   24    0.9397869  0.5226151
##   25    0.9401533  0.5279004
##   26    0.9395981  0.5202700
##   27    0.9399756  0.5294202
##   28    0.9397670  0.5255503
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 6.
plot(rf_model4, main = "\n # Variable selection: Random Forest")

```

## # Variable selection: Random Forest



```
# feature importance
ggplot(varImp(rf_model4), aes(
  x = reorder(feature, Importance),
  y = Importance
)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  theme_classic() +
  labs(
    x = "Feature",
    y = "Importance",
    title = "Feature Importance: Random Forest (Downsampling cv)"
  )
```



```
## manually find the best ntree parameter
tuneGrid4 <- expand.grid(.mtry = 28)
modellist4 <- list()
```

```
# train with different ntree parameters
for (ntree in c(500, 1500, 2500, 5000)) {
  fit <- train(Legendary ~ .,
    data = train_std,
    method = "rf",
    metric = "F",
    tuneGrid = tuneGrid4,
    trControl = train_control_4,
    ntree = ntree
  )
  key <- toString(ntree)
  modellist4[[key]] <- fit
}
```

```
# Compare results
results4 <- resamples(modellist4)
summary(results4)
```

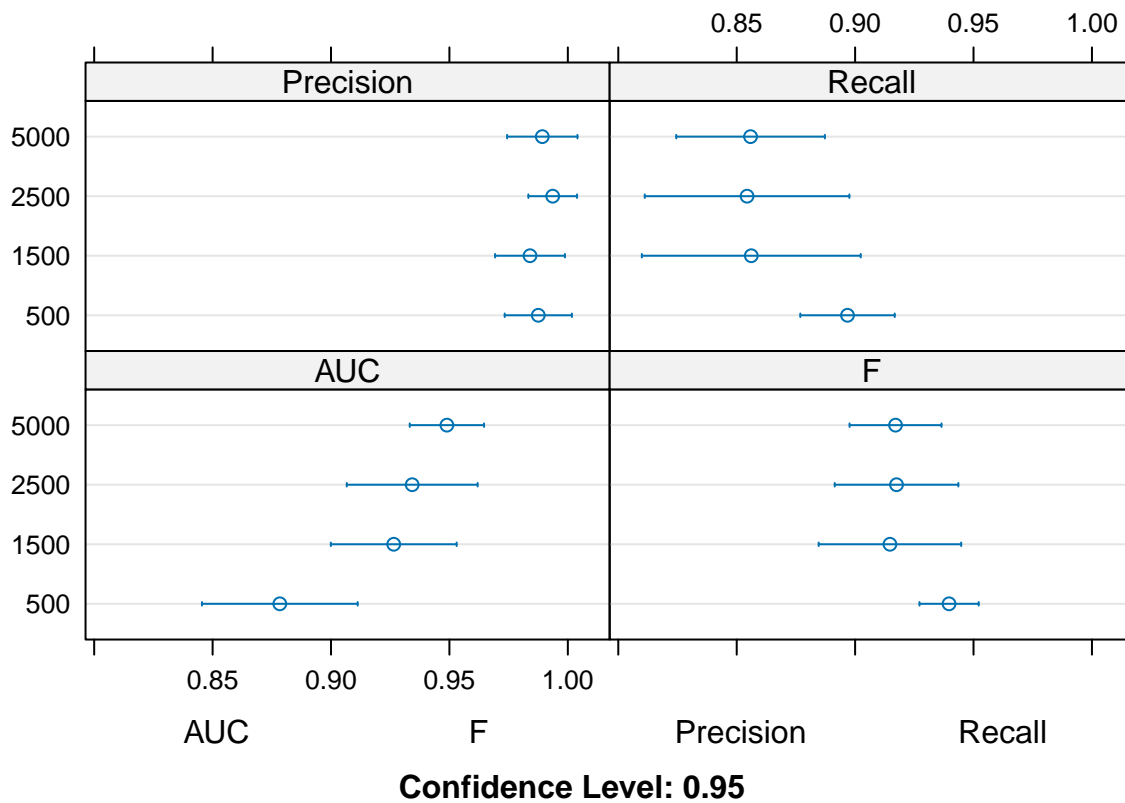
```
##
## Call:
## summary.resamples(object = results4)
##
```

```

## Models: 500, 1500, 2500, 5000
## Number of resamples: 10
##
## AUC
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.7884615 0.8605378 0.8878865 0.8783811 0.8998125 0.9615385    0
## 1500 0.8642640 0.9104089 0.9192220 0.9264831 0.9523893 0.9797106    0
## 2500 0.8393965 0.9240909 0.9386792 0.9342813 0.9586933 0.9796451    0
## 5000 0.9096763 0.9365376 0.9412217 0.9489384 0.9688767 0.9788607    0
##
## F
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.9183673 0.9263158 0.9381443 0.9396751 0.9567615 0.9607843    0
## 1500 0.8602151 0.8823430 0.9073129 0.9147015 0.9537628 0.9702970    0
## 2500 0.8181818 0.9203545 0.9270754 0.9174830 0.9375000 0.9387755    0
## 5000 0.8666667 0.9061428 0.9166667 0.9170654 0.9274552 0.9600000    0
##
## Precision
##           Min.    1st Qu.    Median    Mean 3rd Qu. Max. NA's
## 500  0.9387755 0.9786957 1.0000000 0.9875297      1      1      0
## 1500 0.9361702 0.9757549 0.9883721 0.9840324      1      1      0
## 2500 0.9574468 1.0000000 1.0000000 0.9936170      1      1      0
## 5000 0.9361702 0.9836957 1.0000000 0.9892209      1      1      0
##
## Recall
##           Min.    1st Qu.    Median    Mean    3rd Qu.    Max. NA's
## 500  0.8627451 0.8696267 0.9019608 0.8967949 0.9171380 0.9423077    0
## 1500 0.7692308 0.8048643 0.8544495 0.8561463 0.9117647 0.9423077    0
## 2500 0.6923077 0.8627451 0.8738688 0.8543741 0.8823529 0.9019608    0
## 5000 0.7647059 0.8438914 0.8544495 0.8558446 0.8774510 0.9230769    0
dotplot(results4) # highest Precision-Recall score when ntree=1500, mtry=28

```





selection

on train

accuracy

```
extract_best_accuracy <- function(fit) fit$results[which.max(fit$results$Accuracy), ]
evaluation_rf_accuracy <- as.data.frame(rbind(
  extract_best_accuracy(rf_model1),
  extract_best_accuracy(rf_model2),
  extract_best_accuracy(rf_model3),
  extract_best_accuracy(rf_model4)
))

evaluation_rf_accuracy <- evaluation_rf_accuracy |>
  select(-mtry, -Kappa, -AccuracySD, -KappaSD)
rownames(evaluation_rf_accuracy) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
kable(evaluation_rf_accuracy)
```

	Accuracy
Default-Case	0.9506545
SMOTE	0.9463787
Up-sampling	0.9492380
Down-Sampling	0.9493840

## kappa

```
extract_best_kappa <- function(fit) fit$results[which.max(fit$results$Kappa), ]
evaluation_rf_kappa <- as.data.frame(rbind(
  extract_best_kappa(rf_model1),
  extract_best_kappa(rf_model2),
  extract_best_kappa(rf_model3),
  extract_best_kappa(rf_model4)
))
evaluation_rf_kappa <- evaluation_rf_kappa |>
  select(-mtry, -Accuracy, -AccuracySD, -KappaSD)
rownames(evaluation_rf_kappa) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
kable(evaluation_rf_kappa)
```

	Kappa
Default-Case	0.5847877
SMOTE	0.5821477
Up-sampling	0.5551127
Down-Sampling	0.5478410

## on test

```
rf_models <- list(rf_model1, rf_model2, rf_model3, rf_model4)
# accuracy
get_accuracy <- function(fit) {
  y <- test_std$Legendary
  y_hat <- predict(fit, type = "raw", newdata = test_std |>
    select(-Legendary))
  return(mean(y == y_hat))
}

test_accuracy_rf <- do.call(rbind, lapply(rf_models, get_accuracy))
rownames(test_accuracy_rf) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
colnames(test_accuracy_rf) <- "Accuracy"
kable(test_accuracy_rf) # Up-Sampling
```

	Accuracy
Default-Case	0.9375000
SMOTE	0.9375000
Up-sampling	0.9333333
Down-Sampling	0.9375000

```
## kappa
library(psych)
get_kappa <- function(fit) {
  y <- test_std$Legendary
  y_hat <- predict(fit, type = "raw", newdata = test_std |>
    select(-Legendary))
  return(cohen.kappa(x = cbind(y, y_hat)))
}
test_kappa_rf <- do.call(rbind, lapply(rf_models, get_kappa))
```

```
rownames(test_kappa_rf) <- c("Default-Case", "SMOTE", "Up-sampling", "Down-Sampling")
kable(test_kappa_rf[, 1:2]) # Up-Sampling
```

	kappa	weighted.kappa
Default-Case	0.484536082474227	0.484536082474227
SMOTE	0.452887537993921	0.452887537993921
Up-sampling	0.43379534060749	0.43379534060749
Down-Sampling	0.484536082474227	0.484536082474227

## COMPARING BEST MODELS

```
# install.packages("PRROC")
# library(PRROC)

# plot_precision_recall <- function(true_labels, pred_model1, pred_model2, pred_model3) {
#   pr_model1 <- pr.curve(scores.class0 = pred_model1, weights.class0 = true_labels, curve = TRUE)
#   pr_model2 <- pr.curve(scores.class0 = pred_model2, weights.class0 = true_labels, curve = TRUE)
#   pr_model3 <- pr.curve(scores.class0 = pred_model3, weights.class0 = true_labels, curve = TRUE)
#
#   plot(pr_model1, main = "Precision-Recall Curve Comparison", col = "#1f77b4", lwd = 2)
#   lines(pr_model2, col = "darkblue", lwd = 2)
#   lines(pr_model3, col = "#66b3ff", lwd = 2)
#
#   legend("bottomright",
#         legend = c("Model 1", "Model 2", "Model 3"),
#         col = c("#1f77b4", "darkblue", "#66b3ff"), lwd = 2
#   )
# }

# plot_precision_recall(test$Legendary, logistic_elnet_1, xgb_model2, rf_model4)
```