PH1976 Project: Machine Learning Model for PPD Predictions

Formatted RMarkdown Document

Erin S. King, Jackie Aguilar, Sara Butt, Safa Zia, Lakshmi Kanikkannan

2023-04-25

Contents

Introduction

The following analysis in this PH1976 project is to demonstrate the tools examined in this course for data categorization, regression, and prediction. The project's aim is to predict Parkinson's disease (PD) using the extracted features from the voice recording of patients. For each individual, three recording samples were collected. The data and corresponding analysis is provided by @Sakar in their seminal research on implementing tunable Q-factor wavelet transforms in conjunction with existing data prediction methods. The methods found in this study serve as a roadmap for this project and inform the methods chosen for categorization and prediction.

Definition

Parkinson's disease (PD) is a progressive neuro-degenerative disorder. To accurately detect the disease in the early stage, many tele-diagnosis and tele-monitoring systems have recently been proposed. Since vocal problem is one of the most important symptoms which can be seen in the earlier stage of PD patients, vocal disorders- based systems become popular in PD diagnosis and monitoring. In these systems, various speech signal processing algorithms have been used to extract clinically useful information for PD assessment, and the calculated features are fed to different learning algorithms to make reliable decisions. PD tele-medicine studies showed that the choice of extracted features and learning algorithms directly influences the accuracy and reliability of distinguishing PD patients.

Data

In this study, Sakar et. al collected the voice recordings of 252 subjects including PD patients and healthy individuals. They gathered three recording samples from each subject and extracted seven feature subsets

from the recording samples. The feature subsets were baseline features, intensity-based features, bandwidth and formant features, vocal fold features, Mel Frequency Cepstral Coefficients (MFCC), wavelet transform based features (WT) and tunable Q-factor wavelet transform based features (TQWT).

Study Population

The dataset includes PD patients with age ranging from 33 to 87 (65.1 $\pm$ 10.9) and healthy individuals with age ranging from 41 to 82 (61.1 $\pm$ 8.9). Each patient has three voice recording samples, with 7 aforementioned feature subsets. Each feature subset contains several features.

Methods

Data Partitioning

Instead of using the LOOCV method outlined in the paper, we have decided to break the training dataset into a 90/10 split, where 90% of the data will be used to train, and 10% of the data will be used to check accuracy of the model.

Additionally, the ensemble training set (both **train_df._train** and **train_df._test**) are broken into the following sub feature categories:

- Baseline Features
- Time Frequency Features
  - Intensity based
  - Formant and Bandwidth based
- Vocal Fold Features
- Mel Frequency Cepstral Coefficients (MFCC)
- Wavelet Transform-based Features (WT)
- Tunable Q-Factor Wavelet Transform-based Features (TQWT)

Overall, these seven sub features were used to inform the machine learning model and perform predictions on the test set. The test set is left as full ensemble, and will use the predictions determined from training validation. The best model for each subset will be selected based on these results. Ultimately, an ensemble model consisting of the best predictor for each subset is selected, and weighting is applied based on the accuracy of that model.

```
#Break the training data into feature subsets
train_df.baseline = data.frame(training_data[c(1:24)])
train_df.intensity = data.frame(training_data[c(1:3,25:27)])
train_df.formant = data.frame(training_data[c(1:3,28:35)])
train_df.vff = data.frame(training_data[c(1:3,36:57)])
train_df.mfcc = data.frame(training_data[c(1:3,58:141)])
train_df.wt = data.frame(training_data[c(1:3,142:323)])
train_df.tqwt = data.frame(training_data[c(1:3,324:755)])

#Partition training data
train_indices <- createDataPartition(train_df.baseline$class, p = 0.9, list = FALSE)

# Split train_df data
train_df.baseline_train <- train_df.baseline[train_indices, ]
train_df.baseline_test <- train_df.baseline[-train_indices, ]

train_df.intensity_train <- train_df.intensity[train_indices, ]
train_df.intensity_test <- train_df.intensity[-train_indices, ]

train_df.formant_train <- train_df.formant[train_indices, ]
train_df.formant_test <- train_df.formant[-train_indices, ]

train_df.vff_train <- train_df.vff[train_indices, ]
train_df.vff_test <- train_df.vff[-train_indices, ]

train_df.mfcc_train <- train_df.mfcc[train_indices, ]
train_df.mfcc_test <- train_df.mfcc[-train_indices, ]

train_df.wt_train <- train_df.wt[train_indices, ]
train_df.wt_test <- train_df.wt[-train_indices, ]
```

```
train_df.tqwt_train <- train_df.tqwt[train_indices, ]
train_df.tqwt_test <- train_df.tqwt[-train_indices, ]
```

Standardization

For each feature, the mean and standard deviation of the training set are used to standardize the data, per a traditional Z-score method. This standardization is applied to the **train_df._train**, **train_df._test** and the **test_df** datasets, resulting in data that is normalized with a mean at zero and standard deviation of unity. Using the mean and standard deviation from the training data set ensures that no information leakage occurs from the **train_df._test** or the **test_df** data sets.

```
# Standardizing the data for cross-comparison

require(tidyverse)
require(broom)
require(mosaic)

# Standardizing the data for cross-comparison
# Training and Test Data
subset_names <- c("baseline", "intensity", "formant", "vff", "mfcc", "wt", "tqwt")

# Standardize function
standardize_data <- function(train_df, test_df) {
  for (ii in 4:length(train_df)) {
    mean_val <- mean(train_df[, ii], na.rm = TRUE)
    std_val <- sd(train_df[, ii], na.rm = TRUE)

    train_df[, ii] <- (train_df[, ii] - mean_val) / std_val

    if (!is.null(test_df)) {
      test_df[, ii] <- (test_df[, ii] - mean_val) / std_val
    }
  }
  return(list(train_df, test_df))
}

# Standardize train and test datasets
for (i in subset_names) {
  # Standardize train_df and test_df
  standardized_data <- standardize_data(get(paste0("train_df.", i, "_train")),
↪  get(paste0("train_df.", i, "_test")))
  assign(paste0("train_df_std.", i, "_train"), standardized_data[[1]])
  assign(paste0("train_df_std.", i, "_test"), standardized_data[[2]])
}

# Standardize test dataset
  standardized_data <- standardize_data(get(paste0("training_data")),
↪  get(paste0("test_data")))
  assign(paste0("test_df_std"), standardized_data[[2]])
```

This was accomplished using the **tidyverse**, **broom**, and **mosaic** packages in RStudio. The histograms below shows an example transformation of the original training data set to the standardized form from the **Baseline**, **Intensity**, and **Formant** sub features. Transforming the data allows all data comparisons to be made equivalently. To ensure that training and test data are all benchmarked equivalently, mean and standard deviation is calculated using the training data, and is applied to standardize both the training and test data. This way, no information leakage will occur and the models will be provided standardized data that is unbiased.

The authors considered using PCA analysis to perform data reduction and to minimize multi-colinearity, but this ultimately was decided against for clarity. Due to the inherent complexity that comes along with transforming the data set with PCA, the authors opted to use the standardization method above, and implement a subsequent Random Forest (Boruta) factor selection method following the standardization.

```
plot_subfeatures <- function(subfeature_name, train_df, train_df_std) {
  optimal_mfrow <- function(num_plots) {
    max_cols <- floor(sqrt(num_plots))
    num_rows <- ceiling(num_plots / max_cols)
    return(c(num_rows, max_cols))
  }

  num_plots <- ncol(train_df) - 2

  # Pre-standardization

  par(mfrow = optimal_mfrow(num_plots))
  for (ii in 3:ncol(train_df)) {
    hist(train_df[, ii], main = "", xlab = colnames(train_df)[ii])
  }

  # Post-standardization

  par(mfrow = optimal_mfrow(num_plots))
  for (ii in 3:ncol(train_df_std)) {
    hist(train_df_std[, ii], main = "", xlab = colnames(train_df_std)[ii])
  }
}
```

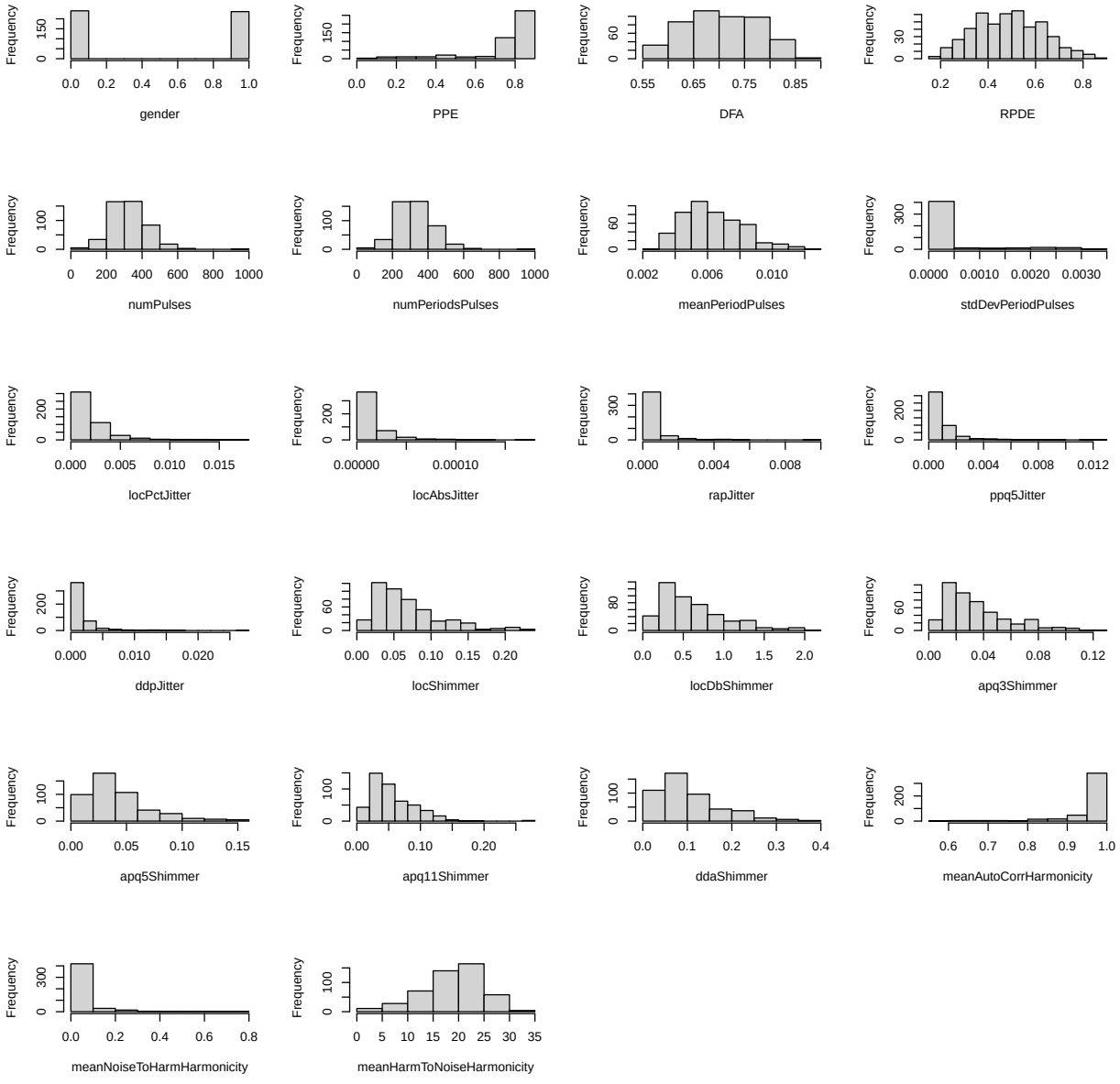

Figure 1: Pre- vs. Post-Standardization Histograms

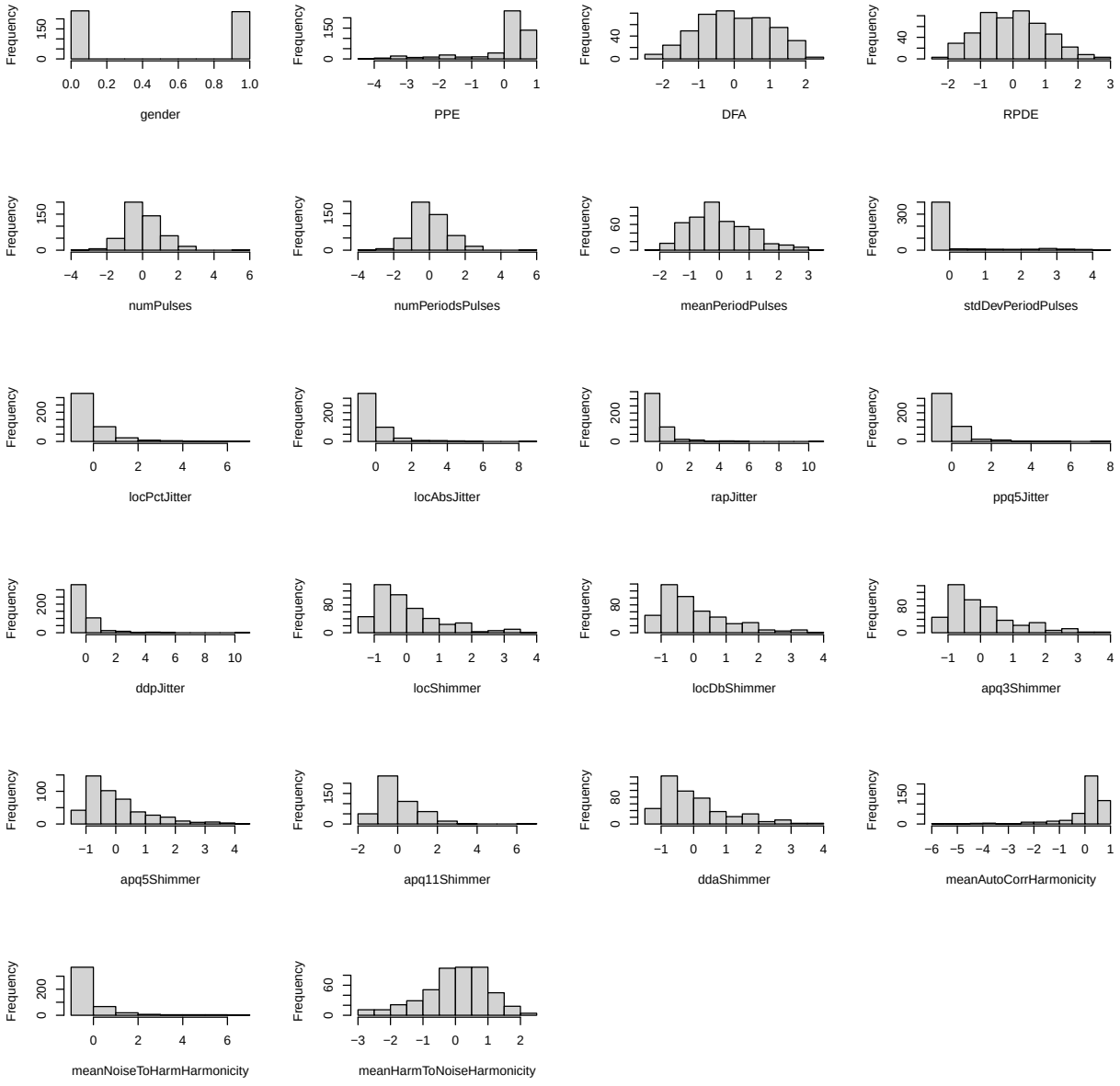

Figure 2: Pre- vs. Post-Standardization Histograms

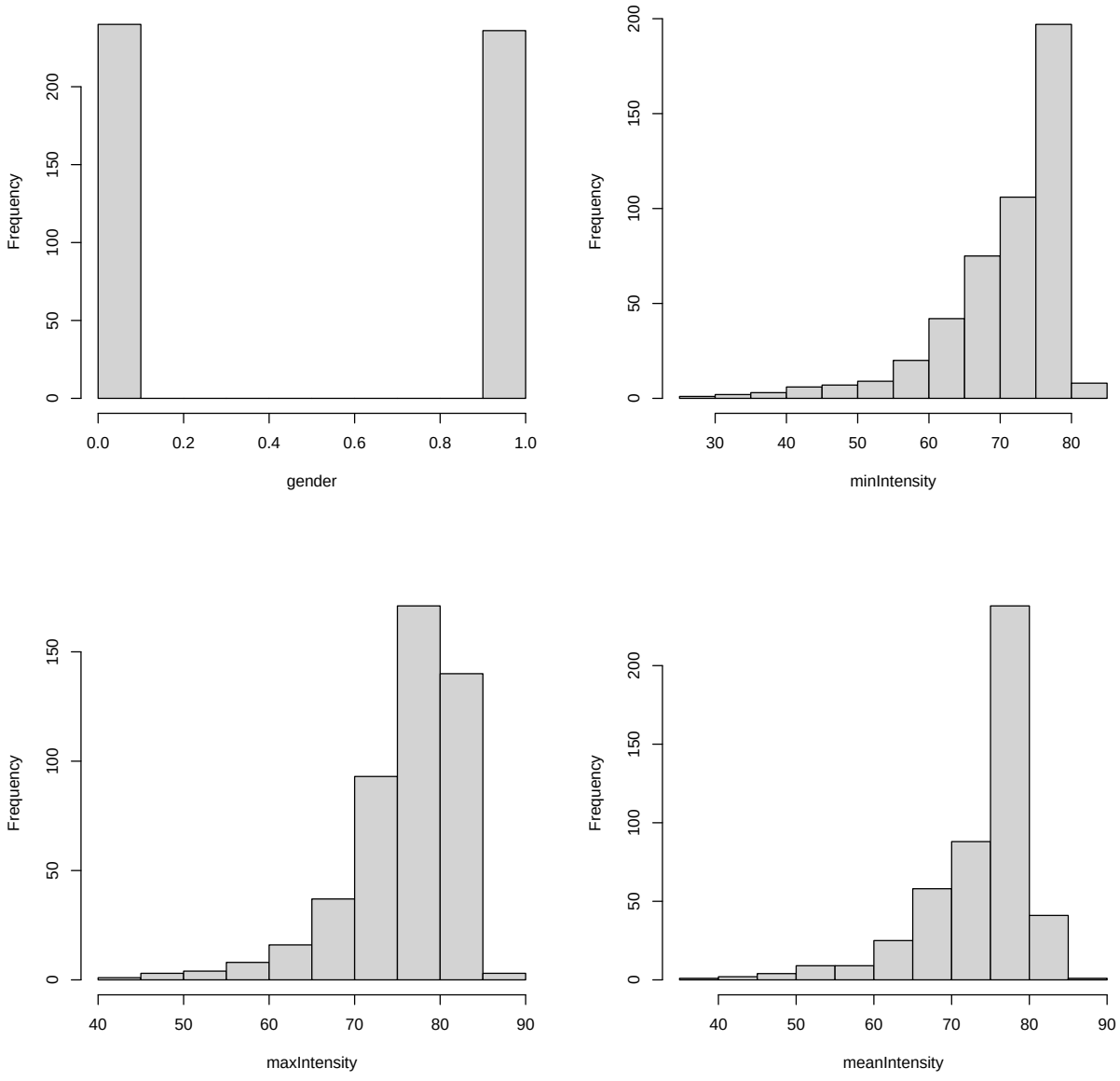

Figure 3: Pre- vs. Post-Standardization Histograms

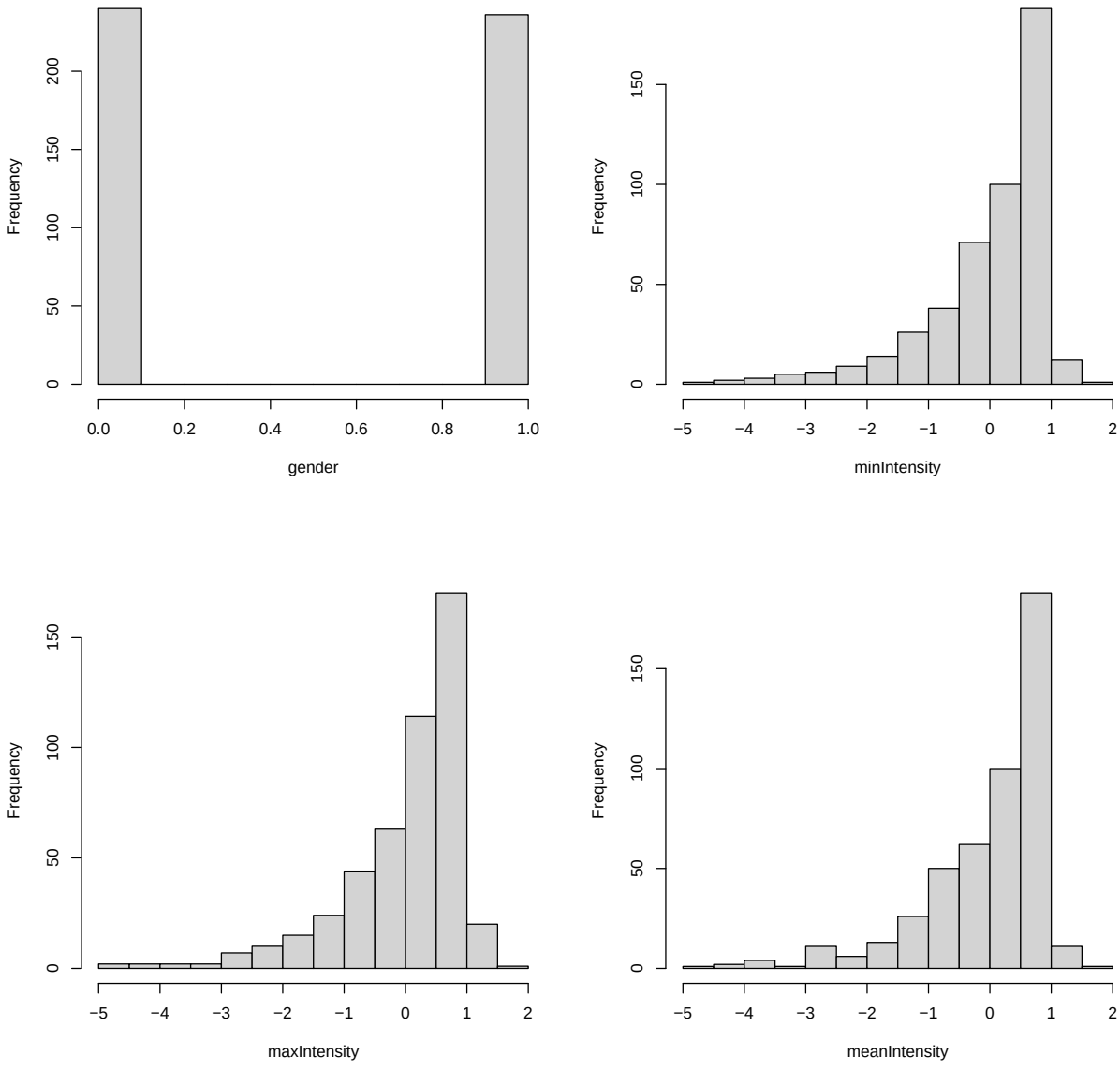

Figure 4: Pre- vs. Post-Standardization Histograms

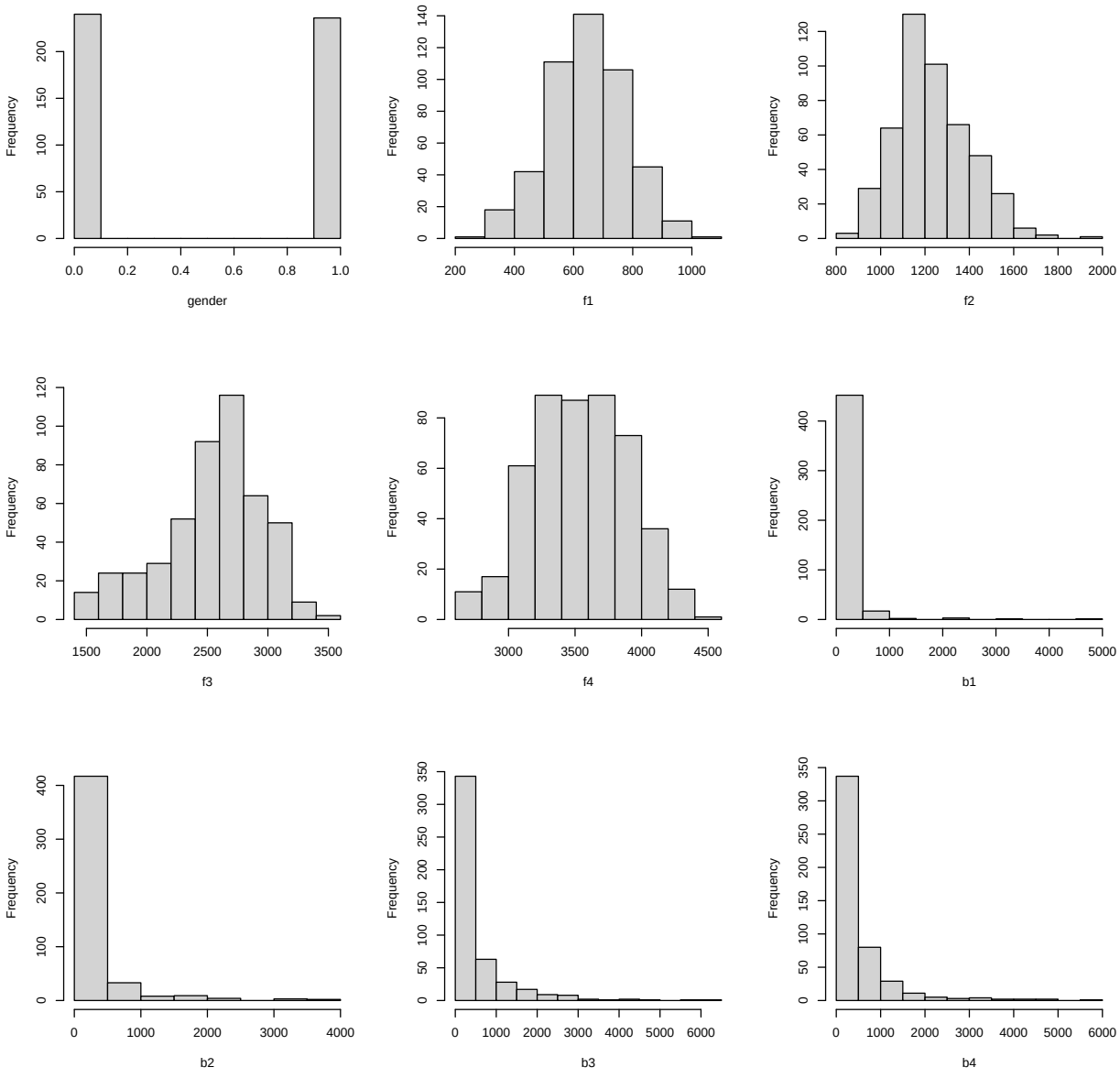

Figure 5: Pre- vs. Post-Standardization Histograms

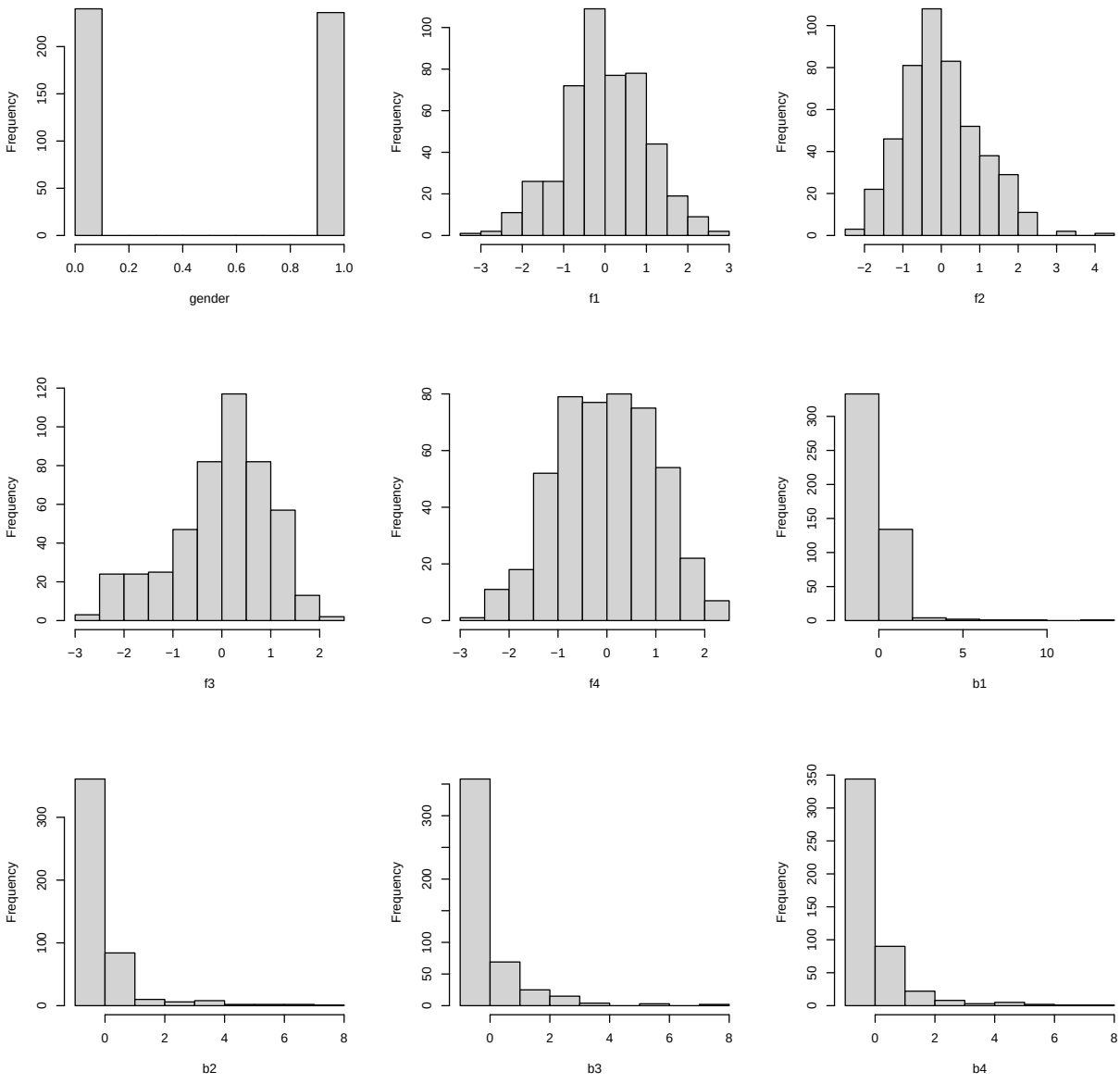

Figure 6: Pre- vs. Post-Standardization Histograms

Feature Selection

Per the **Sakar et al** paper, minimum redundancy-maximum relevance based filter feature selection methods are ideal for determining effective features. The advantage of this is two-fold:
1. It reduces the high dimensionality of the data set. 2. It maximizes the joint dependency of the data set. This strategy is used frequenty in machine learning and regression applications, and as such, will be used in this analysis. The **Boruta** package in RStudio will be used for this purpose, and utilizes Random Forest to perform a top-down search on the corresponding data frame to determine relevant features.

NOTE The Boruta algorithm takes anywhere from 7 to 30 minutes to run, depending on computing power. Please allow the model time to finish.

```
require(Boruta)
require(mlbench)
require(caret)
require(randomForest)
# Function to perform Boruta feature selection
perform_boruta <- function(dataset_name, standardized_train_df, max_runs = 500) {
  cat("Performing Boruta on", dataset_name, "\n")
  set.seed(123)
  boruta_result <- Boruta(class ~ ., data = standardized_train_df, doTrace = 2, maxRuns =
↪  max_runs)
  return(boruta_result)
}

# Call the perform_boruta function for each subset. Finds important features in each
↪  subset
boruta_results <- list()
for (subset_name in subset_names) {
  standardized_train_df <- get(paste0("train_df_std.", subset_name, "_train"))
  boruta_result <- perform_boruta(subset_name, standardized_train_df)
  boruta_results[[subset_name]] <- boruta_result
}
```

mRMR analysis yielded the following results. TQWT results are particularly dense and the plot is somewhat difficult to interpret; however, the overall trend is such that:
- Red regions are categorically rejected and excluded from the included features.
- Yellow regions are tentative, and are handled in a later section of code.
- Green regions are found to be important and thus selected as included features.

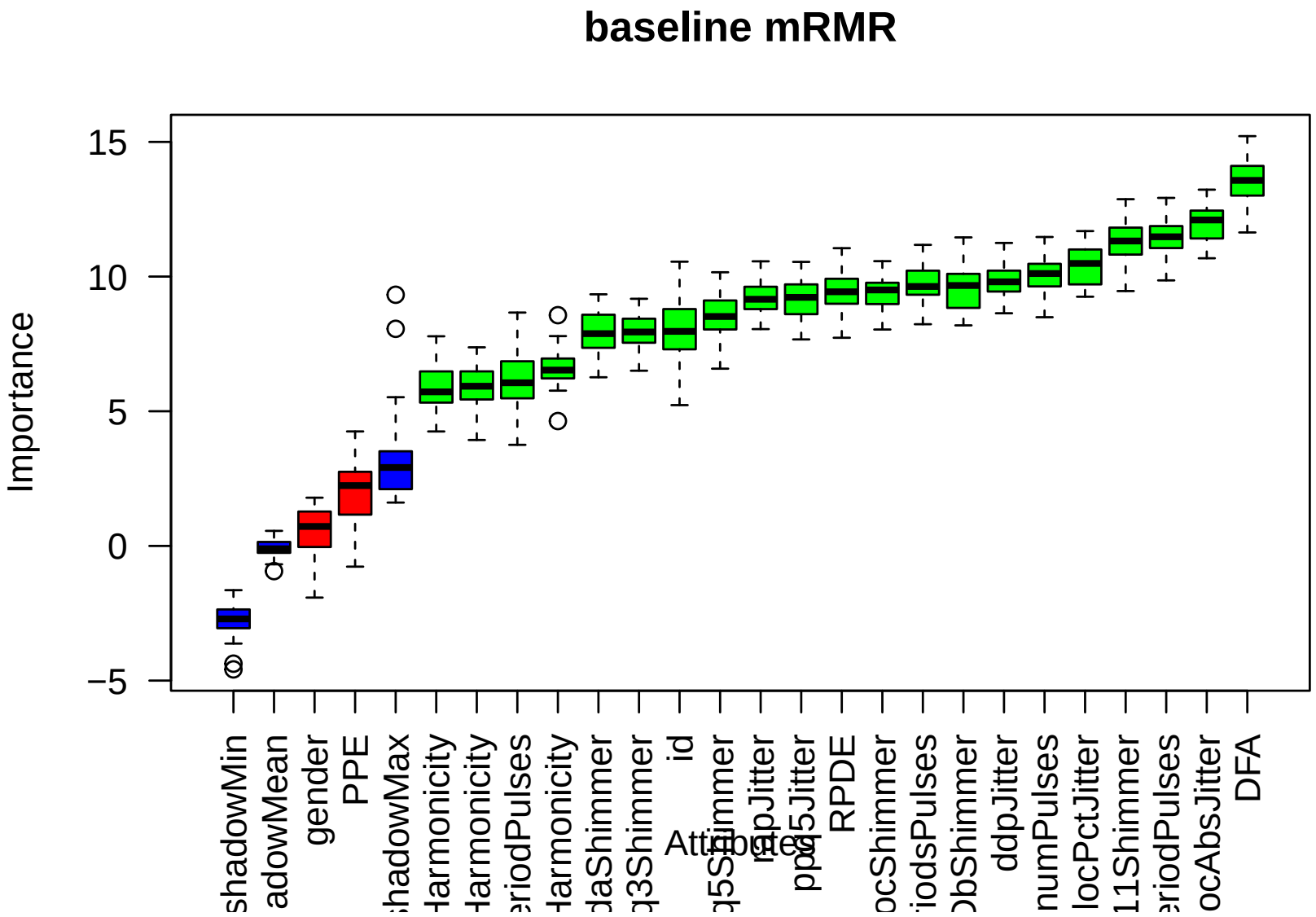

Figure 7: Boruta Plot

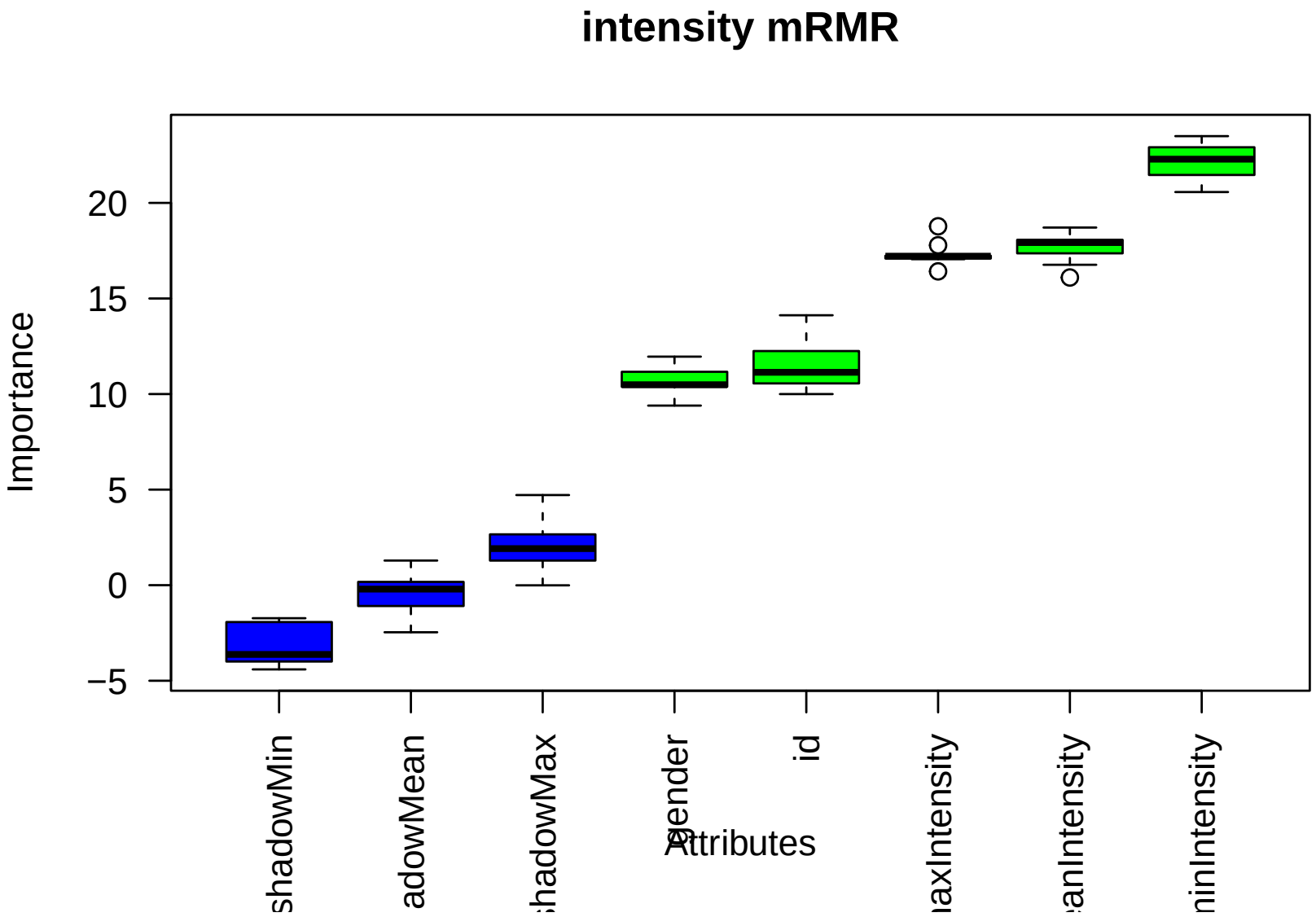

Figure 8: Boruta Plot

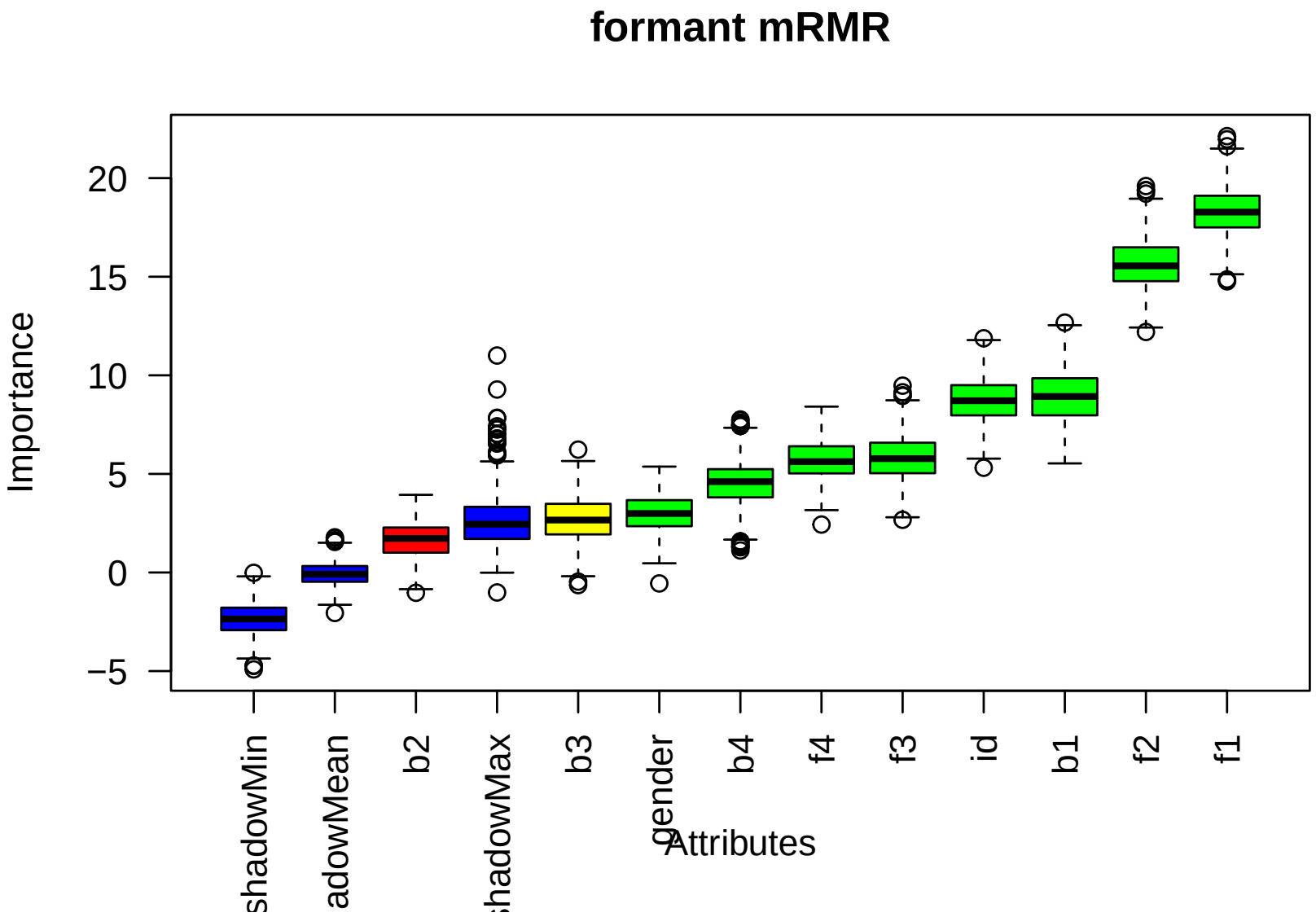

Figure 9: Boruta Plot

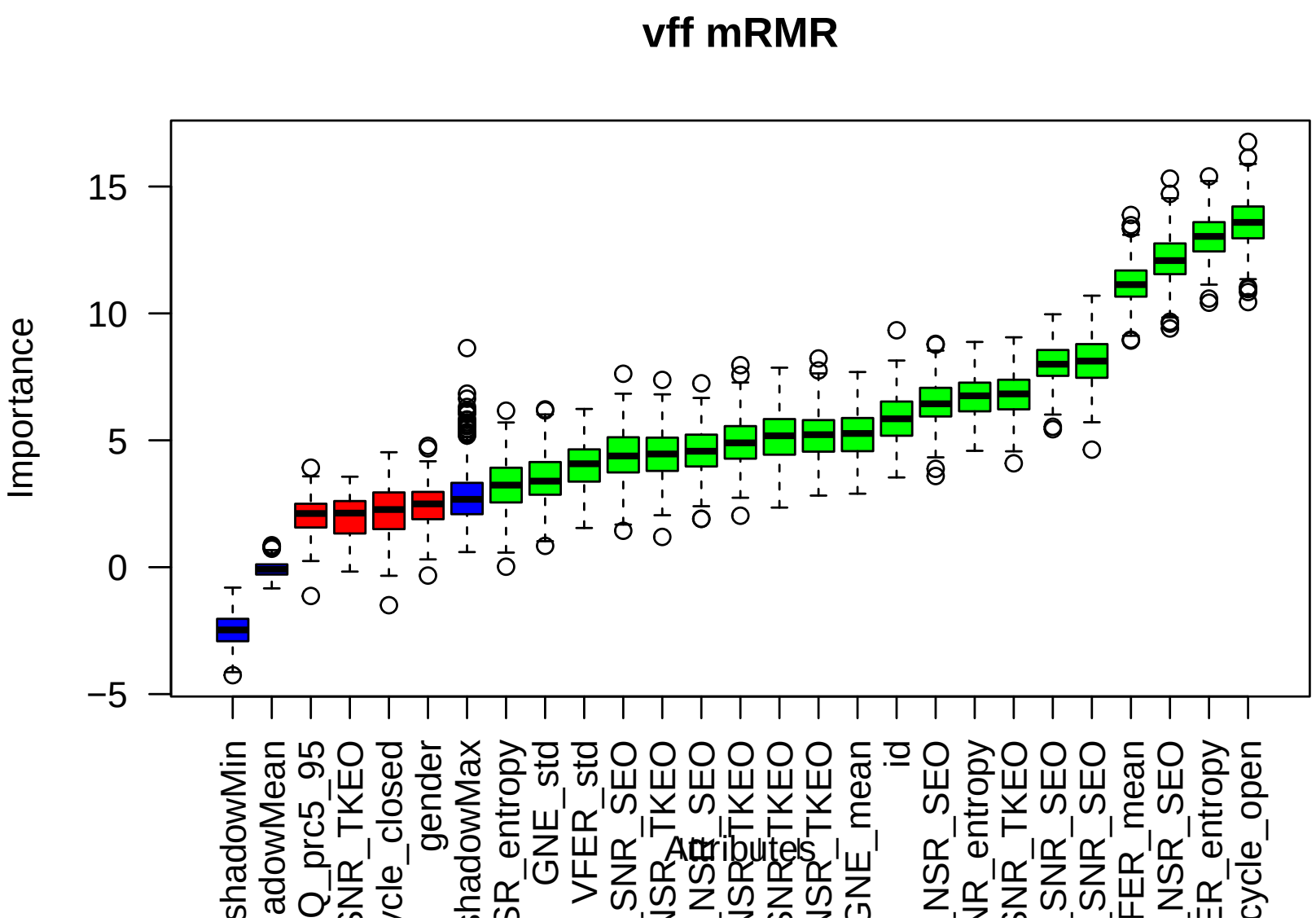

Figure 10: Boruta Plot

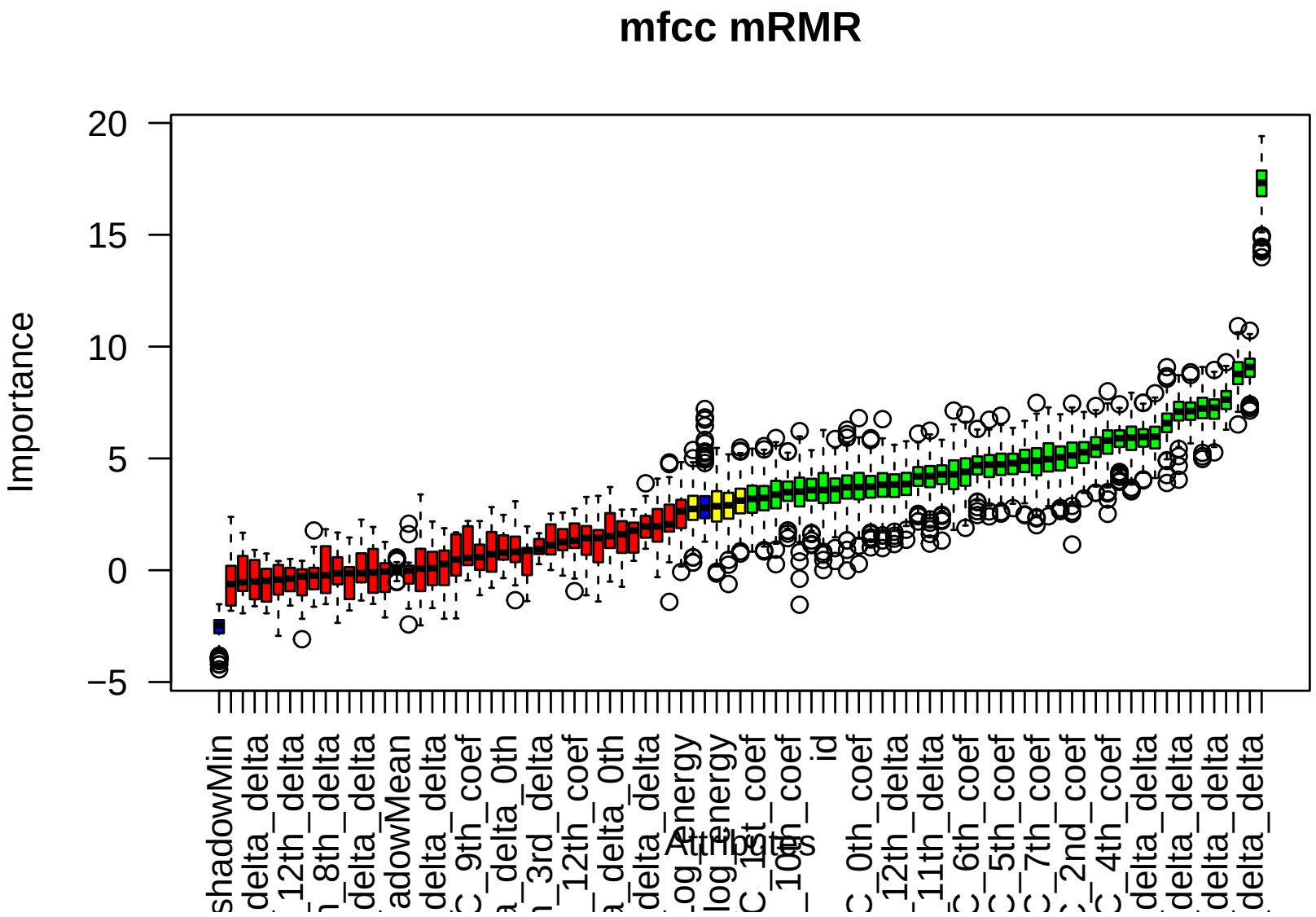

Figure 11: Boruta Plot

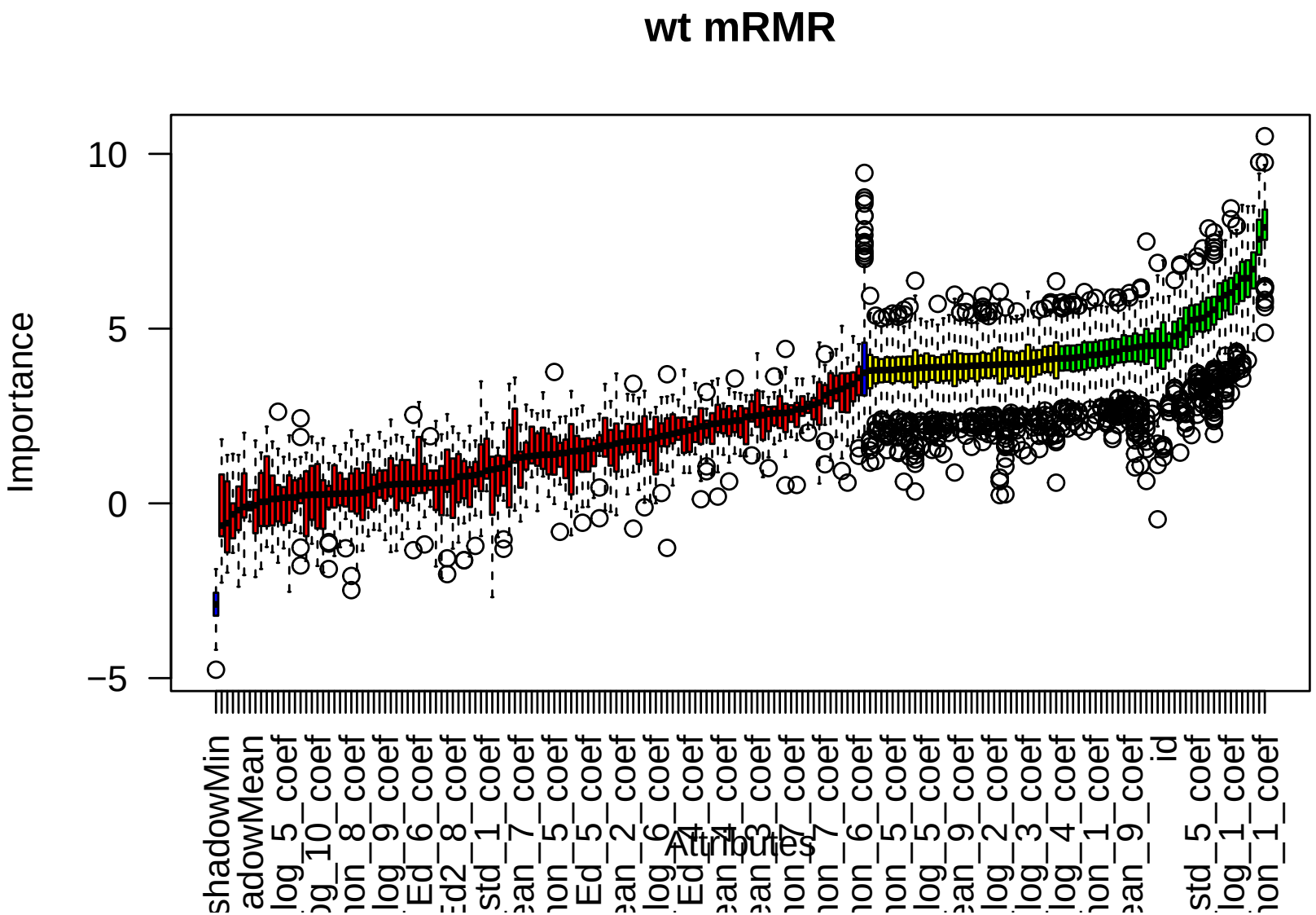

Figure 12: Boruta Plot

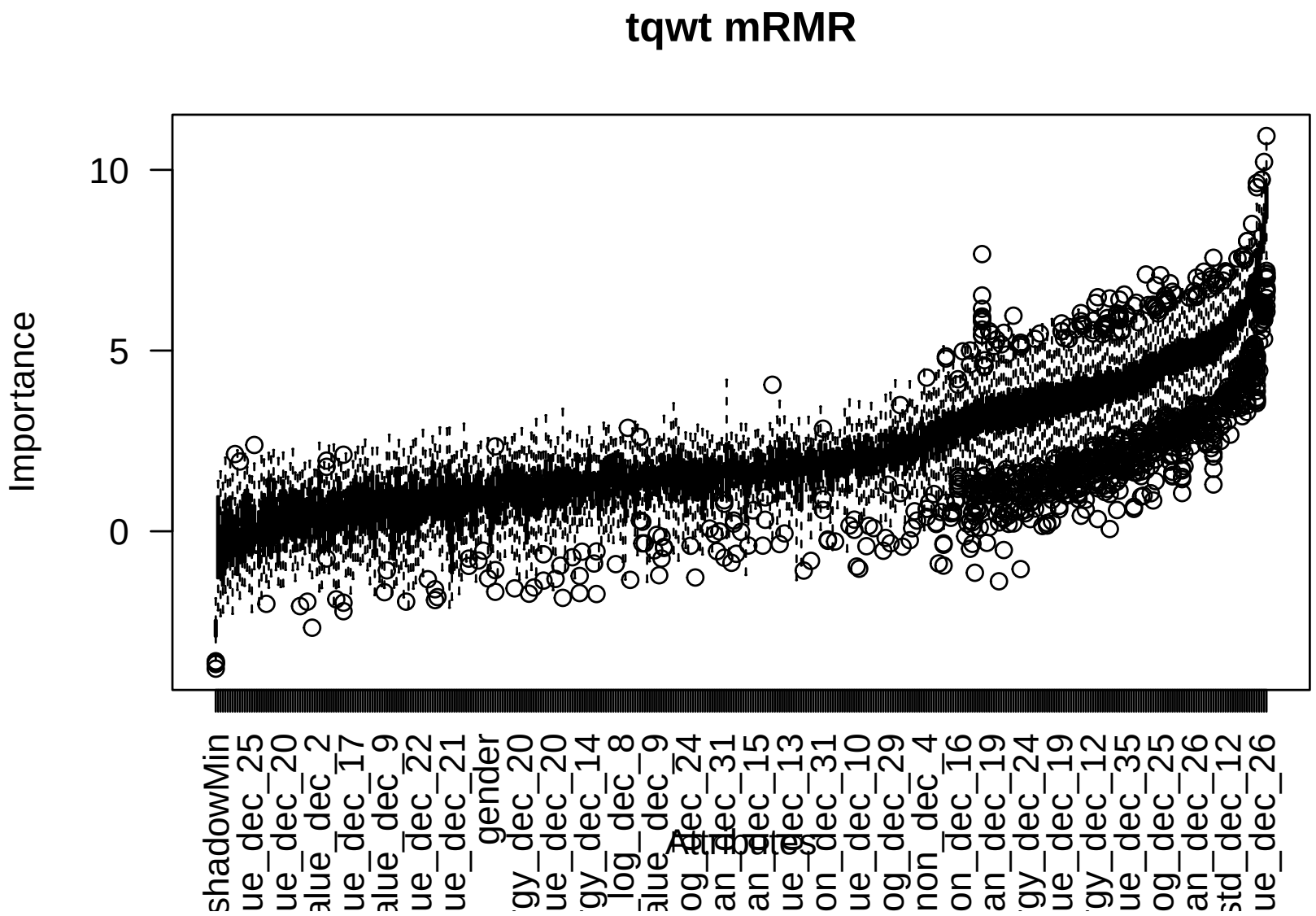

Figure 13: Boruta Plot

Following this initial assessment, chosen variables are selected for regression by using **getNonRejectedFormula()**. This collapses any variables left as **Tentative** factors into either Accepted or Rejected.

```
#Force "Tentative" values function
get_chosen_features <- function(boruta_results) {
  chosen_features <- list()
  for (name in names(boruta_results)) {
    chosen_formula <- getNonRejectedFormula(TentativeRoughFix(boruta_results[[name]]))
    chosen_features[[name]] <- chosen_formula
  }
  return(chosen_features)
}
```

The following factors were found to be important to the model:

Column 1	Column 2	Column 3	Column 4
class	id	DFA	RPDE
numPulses	numPeriodsPulses	meanPeriodPulses	stdDevPeriodPulses
locPctJitter	locAbsJitter	rapJitter	ppq5Jitter
ddpJitter	locShimmer	locDbShimmer	apq3Shimmer
apq5Shimmer	apq11Shimmer	ddaShimmer	meanAutoCorrHarmonicity
meanNoiseToHarmHarmonicity	meanHarmToNoiseHarmonicity	gender	minIntensity
maxIntensity	meanIntensity	f1	f2
f3	f4	b1	b3
b4	GQ_std_cycle_open	GNE_mean	GNE_std
GNE_SNR_TKEO	GNE_SNR_SEO	GNE_NSR_TKEO	GNE_NSR_SEO
VFER_mean	VFER_std	VFER_entropy	VFER_SNR_TKEO
VFER_SNR_SEO	VFER_NSR_TKEO	VFER_NSR_SEO	IMF_SNR_SEO
IMF_SNR_entropy	IMF_NSR_SEO	IMF_NSR_TKEO	IMF_NSR_entropy
mean_MFCC_0th_coef	mean_MFCC_1st_coef	mean_MFCC_2nd_coef	mean_MFCC_3rd_coef
mean_MFCC_4th_coef	mean_MFCC_5th_coef	mean_MFCC_6th_coef	mean_MFCC_7th_coef
mean_delta_log_energy	mean_2nd_delta	std_Log_energy	std_MFCC_1st_coef
std_MFCC_2nd_coef	std_MFCC_3rd_coef	std_MFCC_4th_coef	std_MFCC_5th_coef
std_MFCC_6th_coef	std_MFCC_7th_coef	std_MFCC_8th_coef	std_MFCC_10th_coef
std_MFCC_11th_coef	std_delta_log_energy	std_1st_delta	std_2nd_delta
std_3rd_delta	std_4th_delta	std_5th_delta	std_6th_delta
std_7th_delta	std_8th_delta	std_9th_delta	std_10th_delta
std_11th_delta	std_12th_delta	std_delta_delta_log_energy	std_1st_delta_delta
std_3rd_delta_delta	std_4th_delta_delta	std_5th_delta_delta	std_6th_delta_delta
std_7th_delta_delta	std_8th_delta_delta	std_9th_delta_delta	std_10th_delta_delta
std_11th_delta_delta	std_12th_delta_delta	Ed_1_coef	Ed_2_coef
Ed_3_coef	det_entropy_shannon_3_coef	det_entropy_log_1_coef	det_entropy_log_2_coef
det_entropy_log_3_coef	det_TKEO_mean_1_coef	det_TKEO_std_1_coef	det_TKEO_std_3_coef
app_entropy_shannon_1_coef	app_entropy_shannon_2_coef	app_entropy_shannon_3_coef	app_entropy_shannon_4_coef
app_entropy_shannon_5_coef	app_entropy_shannon_9_coef	app_entropy_log_1_coef	app_entropy_log_2_coef
app_entropy_log_3_coef	app_entropy_log_4_coef	app_entropy_log_5_coef	app_entropy_log_6_coef
app_entropy_log_9_coef	app_entropy_log_10_coef	app_det_TKEO_mean_4_coef	app_det_TKEO_mean_5_coef
app_det_TKEO_mean_8_coef	app_det_TKEO_mean_9_coef	app_det_TKEO_mean_10_coef	app_TKEO_std_5_coef
app_TKEO_std_6_coef	app_TKEO_std_10_coef	Ed2_1_coef	Ed2_2_coef
Ed2_3_coef	det_LT_entropy_shannon_1_coef	det_LT_entropy_shannon_3_coef	det_LT_entropy_log_1_coef
det_LT_entropy_log_3_coef	det_LT_TKEO_mean_1_coef	det_LT_TKEO_mean_3_coef	det_LT_TKEO_std_1_coef
det_LT_TKEO_std_2_coef	det_LT_TKEO_std_3_coef	app_LT_entropy_shannon_1_coef	app_LT_entropy_shannon_2_coef
app_LT_entropy_shannon_3_coef	app_LT_entropy_shannon_4_coef	app_LT_entropy_shannon_5_coef	app_LT_entropy_shannon_6_coef
app_LT_entropy_shannon_8_coef	app_LT_entropy_shannon_10_coef	app_LT_entropy_log_1_coef	app_LT_entropy_log_2_coef
app_LT_entropy_log_3_coef	app_LT_entropy_log_4_coef	app_LT_entropy_log_5_coef	app_LT_entropy_log_6_coef
app_LT_entropy_log_8_coef	app_LT_entropy_log_9_coef	app_LT_entropy_log_10_coef	app_LT_TKEO_mean_8_coef
app_LT_TKEO_mean_9_coef	app_LT_TKEO_mean_10_coef	app_LT_TKEO_std_5_coef	app_LT_TKEO_std_6_coef
app_LT_TKEO_std_7_coef	app_LT_TKEO_std_8_coef	app_LT_TKEO_std_9_coef	app_LT_TKEO_std_10_coef
tqwt_energy_dec_1	tqwt_energy_dec_2	tqwt_energy_dec_6	tqwt_energy_dec_11
tqwt_energy_dec_12	tqwt_energy_dec_18	tqwt_energy_dec_24	tqwt_energy_dec_25
tqwt_energy_dec_26	tqwt_energy_dec_27	tqwt_energy_dec_28	tqwt_energy_dec_33
tqwt_energy_dec_34	tqwt_energy_dec_35	tqwt_entropy_shannon_dec_1	tqwt_entropy_shannon_dec_6
tqwt_entropy_shannon_dec_11	tqwt_entropy_shannon_dec_12	tqwt_entropy_shannon_dec_13	tqwt_entropy_shannon_dec_14
tqwt_entropy_shannon_dec_15	tqwt_entropy_shannon_dec_32	tqwt_entropy_shannon_dec_33	tqwt_entropy_shannon_dec_34
tqwt_entropy_shannon_dec_35	tqwt_entropy_shannon_dec_36	tqwt_entropy_log_dec_1	tqwt_entropy_log_dec_12
tqwt_entropy_log_dec_13	tqwt_entropy_log_dec_16	tqwt_entropy_log_dec_18	tqwt_entropy_log_dec_19
tqwt_entropy_log_dec_25	tqwt_entropy_log_dec_26	tqwt_entropy_log_dec_27	tqwt_entropy_log_dec_28
tqwt_entropy_log_dec_32	tqwt_entropy_log_dec_33	tqwt_entropy_log_dec_34	tqwt_entropy_log_dec_35
tqwt_TKEO_mean_dec_2	tqwt_TKEO_mean_dec_6	tqwt_TKEO_mean_dec_11	tqwt_TKEO_mean_dec_12
tqwt_TKEO_mean_dec_13	tqwt_TKEO_mean_dec_18	tqwt_TKEO_mean_dec_19	tqwt_TKEO_mean_dec_25
tqwt_TKEO_mean_dec_26	tqwt_TKEO_mean_dec_27	tqwt_TKEO_mean_dec_32	tqwt_TKEO_mean_dec_33
tqwt_TKEO_mean_dec_34	tqwt_TKEO_mean_dec_35	tqwt_TKEO_std_dec_6	tqwt_TKEO_std_dec_8
tqwt_TKEO_std_dec_11	tqwt_TKEO_std_dec_12	tqwt_TKEO_std_dec_13	tqwt_TKEO_std_dec_14
tqwt_TKEO_std_dec_17	tqwt_TKEO_std_dec_19	tqwt_TKEO_std_dec_25	tqwt_TKEO_std_dec_26
tqwt_TKEO_std_dec_34	tqwt_medianValue_dec_31	tqwt_medianValue_dec_34	tqwt_meanValue_dec_34
tqwt_meanValue_dec_36	tqwt_stdValue_dec_1	tqwt_stdValue_dec_2	tqwt_stdValue_dec_5
tqwt_stdValue_dec_6	tqwt_stdValue_dec_7	tqwt_stdValue_dec_11	tqwt_stdValue_dec_12
tqwt_stdValue_dec_13	tqwt_stdValue_dec_18	tqwt_stdValue_dec_19	tqwt_stdValue_dec_25
tqwt_stdValue_dec_26	tqwt_stdValue_dec_27	tqwt_stdValue_dec_32	tqwt_stdValue_dec_33
tqwt_stdValue_dec_34	tqwt_stdValue_dec_35	tqwt_minValue_dec_7	tqwt_minValue_dec_11
tqwt_minValue_dec_12	tqwt_minValue_dec_13	tqwt_minValue_dec_14	tqwt_minValue_dec_17
tqwt_maxValue_dec_6	tqwt_maxValue_dec_11	tqwt_maxValue_dec_12	tqwt_maxValue_dec_13
tqwt_maxValue_dec_14	tqwt_maxValue_dec_17	tqwt_skewnessValue_dec_24	tqwt_skewnessValue_dec_25
tqwt_skewnessValue_dec_26	tqwt_skewnessValue_dec_27	tqwt_kurtosisValue_dec_12	tqwt_kurtosisValue_dec_16
tqwt_kurtosisValue_dec_17	tqwt_kurtosisValue_dec_18	tqwt_kurtosisValue_dec_19	tqwt_kurtosisValue_dec_20
tqwt_kurtosisValue_dec_22	tqwt_kurtosisValue_dec_25	tqwt_kurtosisValue_dec_26	tqwt_kurtosisValue_dec_27
tqwt_kurtosisValue_dec_29	tqwt_kurtosisValue_dec_32	tqwt_kurtosisValue_dec_33	tqwt_kurtosisValue_dec_34
tqwt_kurtosisValue_dec_35	NA	NA	NA

Model Selection and Weighting

Once the important features had been determined, they can be used to inform the predictive model for each sub-group. For this analysis, a function was built to test each sub-group against a number of predictive models. Using the 10% "test" data of the training set, accuracy estimates were generated and used to benchmark the model's performance against each other. The models used for analysis were:

- Multilayer Perceptron
- Logistic Regression
- Random Forest
- SVM w/ Linear Kernel
- SVM w/ Radial Kernel
- Naive Bayes
- k- Nearest Neighbors

```
#Function for Multilayer Perceptron
train_mlp <- function(train_df, formula) {
  library(neuralnet)
  set.seed(123)

  threshold_func <- function(x) ifelse(x > 0.5, 1, 0)
  train_df$class <- as.numeric(train_df$class) - 1

  mlp_model <- neuralnet(formula, data = train_df, hidden = c(5), linear.output = FALSE,
↪  act.fct = "logistic", stepmax = 1e+05)
  return(list(model = mlp_model, threshold_func = threshold_func))
}

test_model <- function(model_obj, test_df, model_type, chosen_formula) {
  if (model_type == "mlp") {
    model <- model_obj$model
    threshold_func <- model_obj$threshold_func

    test_data <- model.matrix(chosen_formula, data = test_df)[, -1]
    predictions <- compute(model, test_data)$net.result
    predicted_classes <- sapply(predictions, threshold_func)
    actual_classes <- test_df$class

    accuracy <- sum(predicted_classes == actual_classes) / length(actual_classes)
  } else {
    predictions <- predict(model_obj, test_df)

    if (model_type %in% c("logit", "svm_linear", "svm_rbf")) {
      predicted_classes <- ifelse(predictions > 0.5, 1, 0)
    } else {
      predicted_classes <- predictions
    }
    actual_classes <- test_df$class

    accuracy <- sum(predicted_classes == actual_classes) / length(actual_classes)
```

```
  }

  return(accuracy)
}
```

For this analysis it required the use of the **caret**, **randomForest**, **e1071**, **nnet**, **kernlab**, and **naivebayes** libraries.

```
#Modeling function
generate_models <- function(dataset_name, train_df, test_df) {
  library(caret)
  library(randomForest)
  library(e1071)
  library(nnet)
  library(kernlab)
  library(naivebayes)

  set.seed(123)
  # Create chosen_formula
  chosen_formula <- as.formula(chosen_features[[dataset_name]])

  # Convert the class variable into a factor
  train_df$class <- as.factor(train_df$class)
  test_df$class <- as.factor(test_df$class)

  # Create chosen_formula
  chosen_formula <- as.formula(chosen_features[[dataset_name]])

  # Train/test data
  train_data <- model.matrix(chosen_formula, data = train_df)[, -1]
  train_class <- train_df$class
  test_data <- model.matrix(chosen_formula, data = test_df)[, -1]
  test_class <- test_df$class
  # Initialize list to store models and accuracy
  models_and_accuracy <- list()

  # Logistic Regression
  logit_model <- glm(formula = chosen_formula, family = "binomial", data = train_df)
  logit_predictions <- predict(logit_model, newdata = test_df, type = "response")
  logit_predicted_classes <- ifelse(logit_predictions > 0.5, 1, 0)
  accuracy_logit <- sum(logit_predicted_classes == test_class) / length(test_class)

  models_and_accuracy[["Logistic Regression"]] <- list(model = logit_model, accuracy =
↪  accuracy_logit)

  # Define the parameter grid for tuning the Random Forest
  tuneGrid <- expand.grid(mtry = sqrt(ncol(train_df)),
                          splitrule = "gini",
                          min.node.size = c(1, 3, 5, 10, 15))

  # Set up cross-validation
  cvControl <- trainControl(method = "cv", number = 5, search = "grid")

  # Random Forest model using cross-validation
```

```
  rf_model <- train(chosen_formula, data = train_df,
                    method = "ranger",
                    trControl = cvControl,
                    tuneGrid = tuneGrid,
                    importance = "none",
                    num.trees = 500)

  rf_pred <- predict(rf_model, newdata = test_df)
  accuracy_rf <- sum(rf_pred == test_class) / length(test_class)

  models_and_accuracy[["Random Forest"]] <- list(model = rf_model, accuracy =
↪  accuracy_rf)

  # SVM with Linear Kernel
  svm_linear <- svm(train_data, train_class, kernel = "linear")
  predictions_linear <- predict(svm_linear, test_data)
  accuracy_linear <- sum(predictions_linear == test_class) / length(test_class)

  models_and_accuracy[["SVM Linear"]] <- list(model = svm_linear, accuracy =
↪  accuracy_linear)

  # SVM with RBF Kernel
  svm_rbf <- svm(train_data, train_class, kernel = "radial")
  predictions_rbf <- predict(svm_rbf, test_data)
  accuracy_rbf <- sum(predictions_rbf == test_class) / length(test_class)

  models_and_accuracy[["SVM RBF"]] <- list(model = svm_rbf, accuracy = accuracy_rbf)

   # Multilayer Perceptron
  mlp_model <- train_mlp(train_df, chosen_formula)
  accuracy_mlp <- test_model(mlp_model, test_df, "mlp", chosen_formula)

  models_and_accuracy[["Multilayer Perceptron"]] <- list(model = mlp_model$model,
↪  accuracy = accuracy_mlp)

  # Naive Bayes
  nb_model <- naive_bayes(chosen_formula, data = train_df)
  nb_predictions <- predict(nb_model, newdata = test_df)
  accuracy_nb <- sum(nb_predictions == test_class) / length(test_class)

  models_and_accuracy[["Naive Bayes"]] <- list(model = nb_model, accuracy = accuracy_nb)

  # KNN
  k <- 10
  knn_predictions <- knn(train = train_data, test = test_data, cl = train_class, k = k)
  accuracy_knn <- sum(knn_predictions == test_class) / length(test_class)
  knn_model <- list(train_data = train_data, train_class = train_class, k = k)

  models_and_accuracy[["KNN"]] <- list(model = knn_model, accuracy = accuracy_knn)
  return(models_and_accuracy)
}
```

The following results were found for each of the sub features. A comparative bar chart for each of the sub features is also included.

```
# Initialize the data frame
model_accuracies <- data.frame()
best_models <- list()
# Iterate over the subsets
for (subset_name in subset_names) {
  train_df <- get(paste0("train_df_std.", subset_name, "_train"))
  test_df <- get(paste0("train_df_std.", subset_name, "_test"))
  models_and_accuracy <- suppressWarnings(generate_models(subset_name, train_df,
↪  test_df))

  # Create a data frame to store model accuracies for each subset
  subset_model_accuracies <- data.frame(model = names(models_and_accuracy), accuracy =
↪  unlist(lapply(models_and_accuracy, function(x) x$accuracy)), stringsAsFactors =
↪  FALSE)
  subset_model_accuracies$subset_name <- subset_name

  # Bind the rows to the model_accuracies data frame
  model_accuracies <- rbind(model_accuracies, subset_model_accuracies)

  # Find the model with the highest accuracy
  best_model <- names(which.max(sapply(models_and_accuracy, function(x) x$accuracy)))
  cat("Best model for", subset_name, "is", best_model, "with an accuracy of",
   ↪  models_and_accuracy[[best_model]]$accuracy, "\n")

  # Store the best model for this subset
  best_models[[subset_name]] <- list(name = best_model, model =
↪  models_and_accuracy[[best_model]]$model, accuracy =
↪  models_and_accuracy[[best_model]]$accuracy)
}

# Create the ggplot bar chart
ggplot(data = model_accuracies, aes(x = subset_name, y = accuracy, fill = model)) +
  geom_bar(stat = "identity", position = "dodge") +
  theme_minimal() +
  labs(title = "Model Accuracies by Subset", x = "Subset", y = "Accuracy") +
  scale_fill_brewer(palette = "Set1")
```

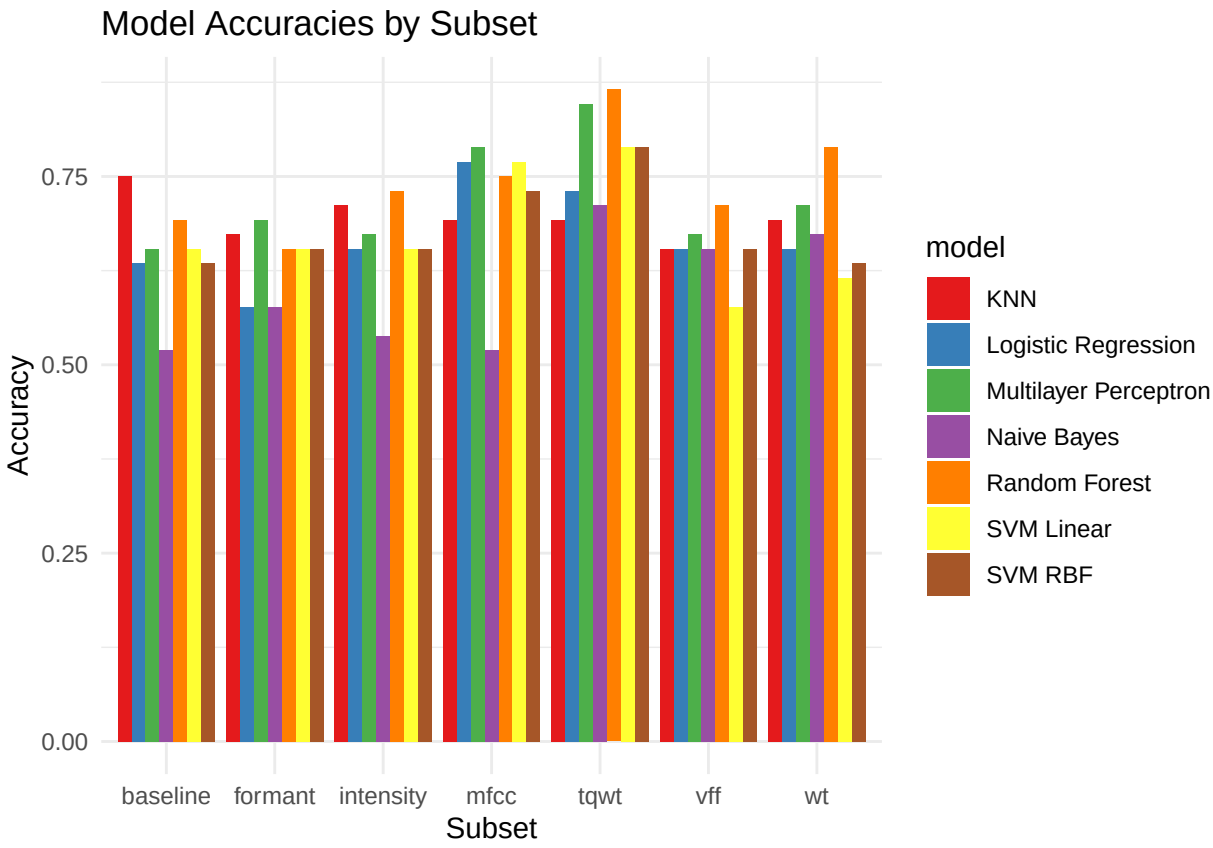

Figure 14: Sub-Feature Results by Algorithm

Ensemble Accuracy Determination

At this point, we have enough data to make an ensemble predictive model, such that the best performing akgorithm for each sub-feature can be used. We used a similar method as the **Sakar et. al** paper, where a combination of weights and predictions are used to develop the ensemble model Y:

$$Y = \sum w_i \cdot d_i \tag{1}$$

$$w_i = \frac{\text{individual model weight}}{\text{sum of all model weights}} \tag{2}$$

$$d_i = prediction_algorithm \tag{3}$$

```
# Functions for Ensemble predictions
weighted_prediction <- function(best_models, test_data, chosen_features) {
  predictions_list <- mapply(function(subset_name, model, test_data, chosen_features) {
    if (inherits(model, "list") && !is.null(model$k)) { # KNN model
      common_columns <- intersect(colnames(test_data[[subset_name]]),
↪  colnames(model$train_data))
  } else if (class(model$finalModel) == "ranger") { # Ranger model
      common_columns <- intersect(colnames(test_data[[subset_name]]),
↪  model$finalModel$forest$independent.variable.names)
  } else if (inherits(model, "nn")) { # Neural Network model
      common_columns <- intersect(colnames(test_data[[subset_name]]),
↪  colnames(model$data))
  } else {
    stop("Unsupported model type")
  }
    test_subset_data <- test_data[[subset_name]][, common_columns]

    if (inherits(model, "glm")) {
      predict(model, newdata = test_subset_data, type = "response")
    } else if (class(model$finalModel) == "ranger") {
      predict(model, newdata = test_subset_data, type = "raw")
    } else if (inherits(model, "svm")) {
      predict(model, newdata = test_subset_data, probability = TRUE)$probabilities[, 2,
      ↪  drop = FALSE]
    } else if (inherits(model, "naiveBayes")) {
      predict(model, newdata = test_subset_data, type = "raw")[, 1, drop = FALSE]
    } else if (inherits(model, "nn")) {
      predictions <- compute(model, test_subset_data)$net.result
      threshold_func <- function(x) ifelse(x > 0.5, 1, 0)
      factor_predictions <- sapply(predictions, threshold_func)
      as.factor(factor_predictions)
    } else if (inherits(model, "list") && !is.null(model$k)) {
      knn(train = model$train_data, test = test_subset_data, cl = model$train_class, k =
      ↪  model$k)
    } else {
      stop("Unsupported model type")
    }
  }, subset_name = names(best_models), model = lapply(best_models, `[[`, "model"),
↪  test_data = rep(list(test_data), length(names(best_models))), chosen_features =
↪  chosen_features, SIMPLIFY = FALSE)
```

```
  # Convert factors to numeric values
  predictions_list <- lapply(predictions_list, function(x) as.numeric(x)-1)

  # Calculate normalized weights
  models_weights <- lapply(best_models, function(x) x$accuracy)
  models_weights_normalized <- unlist(models_weights) / sum(unlist(models_weights))


  # Calculate weighted predictions
  combined_probs <- Reduce(`+`, mapply(`*`, predictions_list, models_weights_normalized,
↪  SIMPLIFY = FALSE))
  combined_predictions <- ifelse(combined_probs > 0.5, 1, 0)

  return(combined_predictions)
}
```

Table 1: Weighted Results and Selected Models per Sub-Feature

	Model	Weights
baseline	KNN	0.1407942
intensity	Random Forest	0.1371841
formant	Multilayer Perceptron	0.1299639
vff	Random Forest	0.1335740
mfcc	Multilayer Perceptron	0.1480144
wt	Random Forest	0.1480144
tqwt	Random Forest	0.1624549

Weights, as shown in Equation 1, are as follows:

ID...1	Result...2	ID...3	Result...4	ID...5	Result...6	ID...7	Result...8
63	1	61	1	67	1	73	1
44	1	10	1	68	0	25	1
29	1	37	1	30	1	48	1
9	1	57	1	15	1	74	1
38	1	8	1	18	1	50	1
59	1	52	1	46	1	33	1
4	1	58	1	19	1	26	1
28	1	31	1	23	1	20	1
62	1	13	1	76	1	72	1
65	1	47	1	43	1	32	1
69	1	36	1	22	1	3	0
39	1	56	1	53	1	51	1
1	1	54	1	27	1	60	1
11	1	70	1	75	1	35	1
40	1	16	1	21	1	34	1
41	1	45	1	71	0	17	1
12	0	24	1	7	1	55	1
64	1	14	0	66	1	2	1
5	1	42	1	49	1	6	1

Results

The results of the ensemble model on test data predictions are found below. These results are also compiled in the **test_results.csv** file.