Applicability of Machine Learning to the TBIMS National Database

CB Eagye, MS, Emily Almeida, MS, Mitch Sevigny, MS

with special thanks to Dr. Eric Munger of the Palo Alto VA Polytrauma Rehabilitation Center

---

Prepared by the
Traumatic Brain Injury Model Systems
National Data and Statistical Center
Craig Hospital

May 2025

This white paper and its associated code were produced with funding from the National Institute on Disability, Independent Living and Rehabilitation Research (NIDILRR), a center within the Administration for Community Living (ACL), Department of Health and Human Services (HHS), under NIDILRR grant # 90DPTB0018, the Traumatic Brain Injury Model Systems National Data and Statistical Center. However, these contents do not necessarily reflect the opinions or views of NIDILRR, ACL, or HHS.

Table of Contents

Introduction

The Traumatic Brain Injury Model Systems (TBIMS) National Data and Statistical Center (NDSC) strives to advance data analysis techniques with respect to the TBIMS National Database (NDB) by continuing to offer a range of analytic options for the TBIMS community to maintain their position at the forefront of rehabilitation research. A principal goal of the NDSC is to recognize when new analytic methods may be applicable for the NDB, how they can be implemented, and ultimately to provide consultation on these methods to interested researchers. Machine learning is one such method. The NDSC has therefore investigated the applicability of machine learning to the NDB as described in this white paper through the examples of three algorithms.

This white paper offers a proof of concept of the utility of machine learning in general to analyze NDB data in future studies and documents the algorithms, code functions, and results using the R coding language. We used standardized R packages for the selected algorithms in order to eliminate decisions and variation intrinsic in writing our own algorithm. This choice is also easily defensible to publication reviewers.

We selected the below three standard machine learning models as examples of classification, deep learning, and regression. The transformation of the raw TBIMS data used with each model is discussed in the first section below, followed by sections for each model that include algorithm definition and features, R code, and results of an example. Note that the R code can be run directly from a package (*TBIMSMachineLearning*) available for download on the NDSC website (https://www.tbindsc.org/StatisticTools.aspx), or it can be copied and pasted into a local copy of RStudio.

Transformation of Data

The first phase of any analysis project is data transformation, also known as “data cleaning,” “data preprocessing,” or “feature engineering”. In the case of traditional machine learning algorithms, the main goal of this transformation process is to improve modeling performance. Moreover, different machine learning models have different requirements, in terms of the type of target or feature engineering that best improves the model performance. In general, feature engineering often includes feature and target selection or filtering, normalization or scaling, appropriate handling of missingness, the need to decide which of similar variables to include in a model, restructuring of categorical data, the potential need for dummy encoding, and other forms of data cleaning.

For the algorithms in this white paper, feature engineering was performed via a series of R functions which are now contained in an R package called *TBIMSDataCleaning* that is also available for download on the NDSC website (https://www.tbindsc.org/StatisticTools.aspx). Users must input their own list of variables. Please note that this package is meant to work on datasets from 2025 and forward.

Please note that the decisions in the subsections below are specific to the goals of the selected algorithms; future researchers should take care to modify the R package or to transform the data to their own specifications if desired.

Handling Categorical Variable Missing Value (8X) Codes

The TBIMS categorical variables have been standardized to ensure consistency in their missing data value codes. These include the following:

- 6-based (e.g., 6, 66, 666, 6666) – Variable Did Not Exist
- 7-based (e.g., 7, 77, 777, 7777) – Refused
- 8-based (e.g., 8, 88, 888, 8888) – Not Applicable
- 9-based (e.g., 9, 99, 999, 9999) – Unknown

For the purposes of our selected algorithms, data cleaning steps included marking variables with certain values as *Available/Not Available* and subsequently coding them as *Missing/Not Missing*.

The 8-based codes in particular pose a problem because they are not strictly coded as 8, 88, 888, and 8888. In certain cases, 8X codes exist (e.g., 81, 87, 89). See the Appendix for details. These variables must be addressed individually based on the purpose of the study or the algorithm chosen in order to decide whether to code them missing or not.

For example, when we performed the initial data cleaning, we were interested in whether someone was *Lost* or *Not Lost* for a random forest survival algorithm and whether their data was *Available* or *Not Available*. The full set of decisions is in the Appendix, but a selection is included here to show the reasoning.

The majority of 8X values for this algorithm represent data that is *Not Available*. However, for a small set of variables this is not true. For example, an IntStatus variable value of 87 has the TBIMS Data Dictionary definition of *Future FollowUp Period*. This value is auto-populated when a Form 2 is created, meaning the subject has not had a follow-up interview yet. For the random forest survival algorithm's purposes, participants with this value were considered *Not Lost*.

In terms of the LostReasonF variable, participants with the TBIMS Data Dictionary value codes of *81 – Not Applicable* and *82 – Not Applicable, Expired* were considered *Not Lost*. Those with *83 – Not Applicable, Funding Not Available* were considered to be *Lost*.

Similarly, a ReasonNoDataIndF value of *81 – Not Applicable, Funding Not Available* was considered *Lost*, but *82 – Not Applicable, Data Was Provided* is considered *Not Lost*.

Another special case example is with the variable DaysTo1stEmp for the value code *88999 – Began Competitive Employment in Prior Follow-Up Year*. Rather than coding this data as *Not Available*, we chose to pull the value from the previous follow-up interview.

These decisions matter when writing code to turn the 8X values into *0 – Lost* or *1 – Not Lost* values. Only the *Not Available* values were coded as 8888; the *Available* codes required a variable-by-variable approach.

Removal of Variables

Most real-world data analysis and modeling projects involve hundreds or thousands of features. In practice, models with many features are more computationally expensive and harder to interpret. Although some algorithms tolerate wide data better than others, including large numbers of low-information predictors will often degrade model performance.

A common example of a low-information predictor is one with little or no variance. Such features typically contribute little to model accuracy. While some algorithms ignore zero-variance predictors, they may still increase computational cost, complicate model deployment, or disrupt resampling procedures, especially when samples contain only the dominant value. In general, low-information predictors are good candidates for removal before model training. For the purposes of this work, TBIMS administrative variables and variables with little or no variance were removed as neither provides information to the algorithms.

After separating all variables into the different types below, an initial seventeen were removed immediately. These included zip codes, death codes, dates, and various administrative variables that were unnecessary to the analysis using any of the models in this paper. Variables were also separated into binary and non-binary, the latter of which could be ordered. Length of Stay (LOS) variables were classified as continuous because of high values for some participants.

Correlated Variables

Including highly-correlated variables in a model risks overfitting and complicates model interpretation. The R data cleaning package includes a function to eliminate the least complete variable of any highly correlated continuous variables. This function provides a Spearman's correlation for each variable with every other variable. Running all continuous variables through it produces their correlation matrix.

The decision of which variables to drop can then be made with input from clinical advisors or data SMEs, based on the correlations. For example, there is a little over 40% missingness in the Income variable ; some researchers prefer to use the Employment variable instead to represent socioeconomic status as it has only 1% missing data. For models in this white paper, Income was dropped and Employment retained.

Categorical Variables

Biostatisticians using TBIMS categorical variables historically restructure them based on the needs of the study. For the purposes of this white paper, categories within a variable were restructured based on cell counts. Within a given variable, some categories are

underrepresented and others are overrepresented. The Appendix contains a list of variables the NDSC suggests considering for this purpose.

Note that the Cause Ecodes were removed from the dataset because they are external (not ICD) cause category codes.

Dummy (Indicator) Variables

Dummy encoding is beneficial for many machine learning algorithms because it transforms categorical variables into a binary format (0/1) that removes any unintended ordinal relationships, allowing models to treat each category as distinct and unrelated. This is especially important for algorithms that rely on distance metrics or linear relationships, such as support vector machines or k-nearest neighbors, which can misinterpret numeric category labels as having an order. In contrast, tree-based models like random forests, gradient boosting machines, and decision trees generally do not require dummy encoding. These algorithms handle categorical variables directly or through label encoding without assuming any ordinal structure.

The R data cleaning package includes a function to take a data frame as input, create the dummy variables, and return the processed data frame. Users should take into consideration the machine learning algorithm being used and weigh the effects of increased dataset width when choosing to dummy encode a predictor.

The Appendix contains a list of variables the NDSC suggests considering for this purpose. If all suggested variables are dummied, the dataset may become too wide and unwieldy, and the resultant model will be very difficult to interpret. For this white paper some categorical variables were collapsed into more condensed categories. These collapsed variables were then run through the R function prior to training distance-based models.

Variables that Change Over Time

The TBIMS NDB contains many variables that change over time, found in the Appendix. Depending on the goal of a study or the requirements of the selected algorithm, such a variable may be included at one or more timepoints or in a different way altogether. Variables must be considered on a case-by-case basis due to the differing uses of each.

For example, TBIMS Form 1 contains FIM variables at both admission and discharge. Assuming a researcher wanted to include only one as a covariate in a model, instead of deciding *a priori* which to include, one strategy is to run three models using FIM at admission (FIMAdmit), FIM at discharge (FIMDischarge), and the calculated change in FIM (DeltaFIM), respectively.

A researcher or clinician can use the results of each to decide whether any was suggestive of clinical importance and should thus be included in the model.

Calculated Variables

For purposes of this white paper, a calculated variable is one that is created via a formula using other variables as inputs. For example, the variable Body Mass Index at Injury (BMI) is calculated using the variables Height (in inches) and Weight (in pounds) as inputs to the formula [Weight/(Height)^2]*703. These types of calculated variables are distinct from the term 'calculated variable' in the TBIMS Data Dictionary, which encompasses both this type as well as any form of manipulation that results in a new variable (e.g., collapsing of categories within a categorical variable).

The Appendix identifies all variables from the TBIMS Data Dictionary that met this white paper's definition of calculated variables. These calculated variables were then eliminated from the model because of correlation risk, while the input variables were retained. However, modeling practitioners could decide to include calculated variables and exclude dependent variables as a strategy for dimensional reduction for model performance improvement.

Imputation

Many machine learning algorithms require complete datasets and cannot handle missing values natively. Algorithms such as logistic regression, support vector machines, and k-nearest neighbors typically fail or return errors when presented with missing values, making imputation a necessary preprocessing consideration. Imputation allows these models to be trained on the full dataset without discarding incomplete rows and may help preserve valuable information. In contrast, some tree-based models such as XGBoost and certain implementations of decision trees can handle missing values internally by learning optimal default directions during tree construction, reducing the need for prior imputation. The decision to impute missing data should be based on adjusting the signal from the predictors in a way that improves model performance. It is misleading to assume that imputation accurately completes a patient record with a value that would have been provided by the patient.

Data imputation methods vary depending on whether the missing values are in categorical or continuous variables. For categorical data, common imputation techniques include replacing missing values with the most frequent category (mode), a constant like "missing," or using predictive models such as decision trees or k-nearest neighbors to estimate the missing category. For continuous data, typical methods include mean or median imputation, interpolation, or model-based approaches like regression or k-nearest neighbors. More advanced techniques for both types include multiple imputation, which accounts for uncertainty by generating several plausible values, and iterative methods such as MICE (Multiple Imputation by Chained Equations), which can model each variable conditionally on the others.

For the support vector machine and artificial neural network algorithms in this white paper, all predictors missing more than 10% were filtered and the remaining predictors (with less than 10% missingness) were imputed. Categorical variables were imputed with the most prevalent

values (the mode), while continuous variables were imputed with means. The R data cleaning package contains these functions.

Note that data were reviewed for missingness and filled in also where it made sense; for example, inserting 'variable did not exist' codes instead of blanks, where applicable.

For this white paper, the missingness review was performed repeatedly as alternate strategies were applied. Plots of missingness are displayed first on the complete set of variables and after each stage after variable removal. This is for demonstration only.

Scaling/Normalization

Data scaling and normalization are important preprocessing steps for many machine learning algorithms because they ensure that features contribute equally to the model, especially when they are on different scales. Algorithms that rely on distance calculations or gradient-based optimization, such as k-nearest neighbors, support vector machines, logistic regression, and neural networks, often perform better with scaled or normalized data, as unscaled features can disproportionately influence the model. In contrast, tree-based algorithms such as decision trees, random forests, and gradient boosting methods (e.g., XGBoost) are generally insensitive to feature scaling, as they split data based on feature thresholds rather than distances or gradients.

For example, many TBIMS indicator codes are intentionally one order of magnitude greater than the maximum response option values (e.g., for TransModeF 99 = Unknown and the response options are 1, 2, 3, or 4 only). The inclusion of such indicator codes may require the predictor to be scaled or normalized prior to training the model. For the algorithms in this white paper, indicator codes were removed and the values were treated as missing.

For example, all 9-based codes (e.g., 9, 99, 999, and 9999) were run through the R function and changed to 9999. Similarly, all 6-based and 7-based codes were changed to be their highest level. Special attention needed to be paid to 8-based codes as described above in the *Handling Categorical Variable Missing Value (8X) Codes* section.

In the case of scaling and normalization, however, care is taken to place the special codes on an ordinal scale and must be done on a case-by-case basis. For example, for the TransModeF variable, a code of *82 – Not Applicable: No Motorized Transportation* does not necessarily indicate no transportation whatsoever. If a researcher wanted to put the information contained in this value code on the ordinal scale, they could either choose to recode it as *0 – No Motorized Transportation* or as *5 – Non-Motorized Transportation*.

The R data cleaning package does not include functions for scaling and normalization as we used extant R functions rather than writing our own.

Dates

Working with dates in machine learning requires more attention than working with other object class types. To prepare a Date object class variable for machine learning, meaningful numerical or categorical features must be extracted from the raw date values, as most algorithms cannot work directly with date objects. Commonly extracted components include the year, month, day, day of the week, hour (if time is included), and whether the date falls on a weekend or holiday. Computing elapsed time features, such as the number of days since a reference date or between events, may be included in a training set. Depending on the problem, cyclical transformations (e.g., using sine and cosine) may be useful for capturing periodic patterns in features like month or day of the week. These derived features can then be used in place of the original date variable, but users are warned that the resulting model will require the same data type transformation to predict new data.

For this white paper, predictors of object class Date were removed. It is also important to note that, for computational efficiency, a limited selection of only a few variables were included in the training of the examples. As a result, none of our examples include special case variables such as imaging data.

Model Selection and Development

The three models below represent standard machine learning algorithms for classification, deep learning, and regression problems.

Model 1: Support Vector Machine

Definition

The support vector machine (SVM) model is a widely-used supervised learning method for classification problems, which involves defining boundaries for separating different classes within a variable. Note that the SVM algorithm cannot handle missing data.

The basic strategy is to optimize a “hyperplane” in the feature space that separates the classes. In real world applications, a hyperplane that perfectly separates the classes isn’t always practical. SVMs address this by accepting boundaries that best – not necessarily perfectly – separate the classes and by using kernel functions to implicitly map input non-linear data into a higher-dimensional space where the data becomes linearly separable. The latter technique is referred to as the “kernel trick.”

To illustrate the model for this white paper, we selected the Satisfaction With Life Scale (SWLS) as the variable to be classified because it is normally distributed and easy to interpret. [1] Variables that are not normally distributed must be transformed prior to running the model. The specific variable name is SWLSTotF, which is the total SWLS score at follow-up (Form 2). We further decided to use SWLSTotF only at the Year 2 follow-up for purposes of this example.

The SWLS data consists of responses to the measure's five items, each scored on a 7-point Likert scale. The total score is calculated by summing the scores for each of the five items, resulting in a total score ranging from 5 to 35, with higher scores indicating greater satisfaction with life. The accepted ranges and cutoff scores are as follows:

31-35: Extremely satisfied
26-30: Satisfied
21-25: Slightly satisfied
20: Neutral
15-19: Slightly dissatisfied
10-14: Dissatisfied
5-9: Extremely dissatisfied

We defined two classes for our example. The first class contains subjects who are "extremely dissatisfied," meaning individuals with SWLSTotF scores of <10. The second class contains all others. We then ran the SVM algorithm using the following subset of variables to create the classifications of SWLS values at Form 2, Year 2:

Days in Post-Traumatic Amnesia (PTADays)
Time to Follow Commands (TFCDays)
Total Length of Stay (LOSTot)
Days from Injury to Rehab Admission (DAYStoREHABadm)
Days from Injury to Rehab Discharge (DAYStoREHABdc)
Marital status (MarF)
Participation (PARTSocialF, PARTProductivityF, PARTOutAboutF, PARTSummaryF)
Anxiety (GAD7TOTF)
Depression (PHQ9TOTF)

While we could have chosen the entire set of TBIMS variables as our predictors, we limited our selection to save run time for the model. With enough time and computing power, we could

[1] Note: It would be relatively easy to swap this variable out with another normally distributed one. The data cleaning has already been done using the R package on the NDSC website (first checking to make sure the data cleaning decisions are suitable to your specific analysis). The algorithm code would only need to have the classification variable swapped out, which lends itself to rapid analysis.

make the set of predictors as large as we want. Readers of this white paper can substitute their own predictors into the code if desired.

Key Components/Steps

Below is the code used to set up and run the SVM model, including code comments. Readers of this white paper are free to copy the code into their own R environment and use it as written or revise it for their own needs.

Step 1: Load data

```
load("F1F2.RData")
```

User Function to set up datasets.

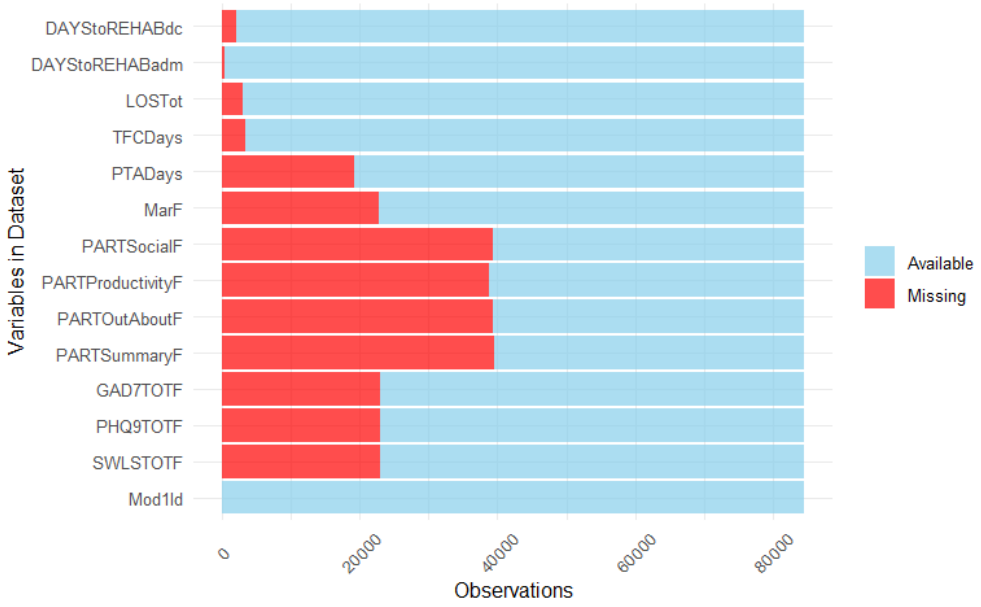
```
select_columns <- function(df, vars) {
  # Check if all specified variables exist in the dataframe
  missing_vars <- setdiff(vars, names(df))

  if (length(missing_vars) > 0) {
    warning("The following variables are not in the dataframe: ", paste(missi
ng_vars, collapse = ", "))
  }

  # Select only the existing columns
  df.Selected <- df[, vars[vars %in% names(df)], drop = FALSE]

  return(df.Selected)
}

# Example usage:
#data <- data.frame(A = 1:5, B = 6:10, C = 11:15)
#vars <- c("A", "C")
#new_df <- select_columns(data, vars)
#print(new_df)
############################################################
```

Step 2: Build data for SWLS

```
library(TBIMS)
library(dplyr)
library(tidyverse)

vars <- c("Mod1Id","PTADays", "TFCDays", "LOSTot", "DAYStoREHABadm", "DAYStoR
EHABdc")
df.SWLSF1 <- select_columns(df.Form1, vars)

vars <- c("Mod1Id","SWLSTOTF", "PHQ9TOTF", "GAD7TOTF", "PARTSummaryF", "PARTO
utAboutF",
          "PARTProductivityF", "PARTSocialF", "MarF")
```

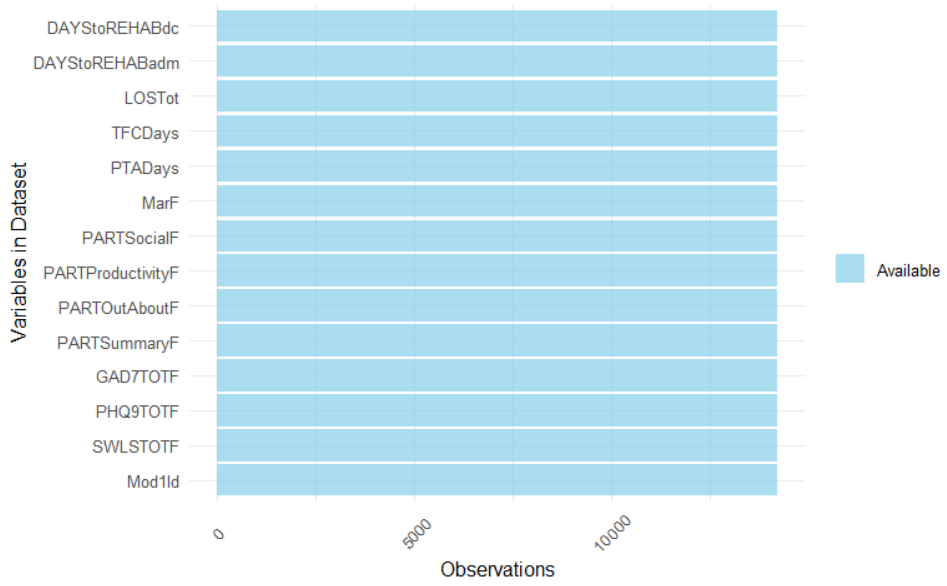
```
df.SWLSF2 <- select_columns(df.Form2, vars)

df.SWLS  <- Combine_dataframes_ID(df.SWLSF2, df.SWLSF1, "Mod1Id")
TBIMS::Missingness_Barplot(df.SWLS)
```

The bar plot output from the code shows missingness for each of the variables we selected for the model. There are roughly 80,000 observations initially. We ran the missingness report repeatedly to visualize the effect of the choices we made.



We decided to ignore all cases with missing data and used the following code to remove them. This reduced our number of observations to around 20,000 as can be seen in the final missingness bar plot.

```
df.SWLS_3 <- df.SWLS_2[complete.cases(df.SWLS_2), ]
TBIMS::Missingness_Barplot(df.SWLS_3)
```



Step 3: Create the SWLS Categories

First we created the standard SWLS ranges referenced above.

```
# This function bins data from a specified column in a dataframe based on val
ue ranges provided.
#
# PARAMETERS:
#   - df: The input dataframe.
#     TYPE: dataframe
#   - column_name: The name of the column in the dataframe to bin.
#     TYPE: character
#   - value_ranges: A vector containing the value ranges to bin the data.
#     TYPE: numeric vector
#     EXAMPLE INPUT: c(0, 10, 20, 30)
# RETURNS:
#   - The input dataframe with a new column added containing the binned value
s.
#     TYPE: dataframe
# EXCEPTIONS:
#   - This function assumes the dataframe contains the specified column and t
he value ranges are in ascending order.
#   - It does not handle cases where the value to bin is outside the specifie
d ranges.
# Example Usage:
#   - Create a sample dataframe
#data <- data.frame(id = 1:5, values = c(5, 15, 25, 10, 20))
#   - Define the value ranges for binning
#ranges <- c(0, 10, 20, 30)
#   - Call the function to bin the 'values' column in the dataframe
#result_df <- bin_data(data, "values", ranges)
```

```
bin_data <- function(df, column_name, value_ranges) {
  # Create a new column to store the binned values
  df$bin_column <- cut(df[[column_name]], breaks = value_ranges, labels = FAL
SE, include.lowest = TRUE)
  return(df)
}
```

We subsequently created the two classes: Extremely Dissatisfied and Not Extremely Dissatisfied.

```
ranges <- c(0, 9, 35)
df.SWLS_4 <- bin_data(df.SWLS_3, "SWLSTOTF", ranges)
#df.SWLS_4 <- df.SWLS_4[0:(nrow(df.SWLS_4) - 17000 ),] # Used to sample the d
ata
```

Step 4: Run the model

In this model, we focused on the following steps:

- Ensure all predictors and the response are numeric
- Scale the predictor features
- Split the data into an 80/20 train-test split
- Check for class imbalance and apply SMOTE if necessary
- Tune SVM hyperparameters using a radial basis function kernel
- Use cross-validation to select the best parameters

```
library(e1071)
library(caret)
library(DMwR)
library(DMwR2)
library(ggplot2)

train_svm <- function(data, predictors, response) {
  # Ensure response variable is a factor
  data[[response]] <- as.factor(data[[response]])

  # Scale predictor variables
  data[, predictors] <- scale(data[, predictors])

  # Split the data into training (80%) and testing (20%)
  set.seed(123)  # For reproducibility
  train_index <- createDataPartition(data[[response]], p = 0.8, list = FALSE)
  train_data <- data[train_index, ]
  test_data <- data[-train_index, ]

  # Ensure test set response variable has the same factor levels as the
training set
  test_data[[response]] <- factor(test_data[[response]], levels =
levels(train_data[[response]]))
```

```
  # Check for class imbalance and apply SMOTE if necessary
  class_counts <- table(train_data[[response]])
  min_class <- min(class_counts)
  if (min_class / max(class_counts) < 0.5) {  # Imbalance threshold
    message("Imbalanced Target - Applying SMOTE")
    train_data <- SMOTE(as.formula(paste(response, "~", paste(predictors,
collapse = "+"))), data = train_data)
  }

  # Define parameter tuning grid
  tune_grid <- expand.grid(
    C = 2^(-5:5),  # Regularization parameter
    sigma = 2^(-5:5)  # RBF kernel parameter
  )

  # Train SVM with hyperparameter tuning
  svm_model <- train(
    as.formula(paste(response, "~", paste(predictors, collapse = "+"))),
    data = train_data,
    method = "svmRadial",
    trControl = trainControl(method = "cv", number = 5),  # 5-fold cross-
validation
    tuneGrid = tune_grid
  )

  # Predict on test data
  predictions <- predict(svm_model, test_data)

  # Ensure predictions are factors with the same levels as the actual
response
  predictions <- factor(predictions, levels = levels(test_data[[response]]))

  # Compute performance metrics
  confusion <- confusionMatrix(predictions, test_data[[response]])
  accuracy <- confusion$overall["Accuracy"]

  # Extract per-class metrics correctly
  if (is.matrix(confusion$byClass)) {
    precision <- confusion$byClass[, "Precision"]
    recall <- confusion$byClass[, "Recall"]
  } else {
    precision <- confusion$byClass["Precision"]
    recall <- confusion$byClass["Recall"]
  }

  # Compute F1-score
  f1_score <- 2 * (precision * recall) / (precision + recall)

  # Aggregate multi-class scores (macro-average)
  macro_precision <- mean(precision, na.rm = TRUE)
  macro_recall <- mean(recall, na.rm = TRUE)
  macro_f1 <- mean(f1_score, na.rm = TRUE)
```

```
  # Generate performance plots
  plot_data <- data.frame(True = test_data[[response]], Predicted =
predictions)
  plot <- ggplot(plot_data, aes(x = True, y = Predicted)) +
    geom_jitter(width = 0.2, height = 0.2, alpha = 0.5) +
    labs(title = "SVM Predictions vs True Values", x = "True Values", y =
"Predicted Values") +
    theme_minimal()

  return(list(
    model = svm_model,
    test_data = test_data,
    performance = list(
      accuracy = accuracy,
      precision = macro_precision,
      recall = macro_recall,
      f1_score = macro_f1
    ),
    plot = plot
  ))
}

############################################################

set.seed(42)
# Define predictors and response
predictors <- colnames(df.SWLS_4)[3:(ncol(df.SWLS_4) - 1)]
response <- "bin_column"
#df.SWLS_4$bin_column <- as.factor(df.SWLS_4$bin_column)

# Train the SVM model
svm_results <- train_svm(df.SWLS_4, predictors, response)
Imbalanced Target - Applying SMOTE
```

Model Fit and Results

Accuracy, precision, recall, and F1 score are all metrics used to measure how accurate a model is. Accuracy is the percentage of correctly classified cases. Precision represents the ratio of correctly classified cases *of those in the primary class (Extremely Dissatisfied)* to the total predicted to be in this class. Recall is the ratio of those in the primary class to the total who are truly in the class. And F1 score is a combination of precision and recall that balances the two metrics.

Our SVM model resulted in the following performance metric values:

Accuracy = 0.8845883
Precision = 0.2680412
Recall = 0.2184874
F1 Score = 0.2407407

Additionally, we generated a plot of predicted versus true values using the test data that was separated from the training data above. This type of plot is called a *confusion matrix* and is another way to assess the performance of the SVM and other classification algorithms. To produce a confusion matrix the actual values must be known, which they are in our case.

In the figure below, 1 = Not Extremely Dissatisfied and 2 = Extremely Dissatisfied.

Confusion matrices are essentially cross-tabulations which can be represented with counts in each quadrant or, as in our figure, with data points. The 2,2 quadrant in the top right represents those cases who are actually Extremely Dissatisfied and who were predicted to be Extremely Dissatisfied. The 1,1 quadrant in the bottom left represents those who were truly Not Extremely Dissatisfied and were predicted correctly. The 1,2 and 2,1 quadrants represent those cases who were incorrectly predicted, known as Type 1 and Type 2 errors.

Researchers who run this code can add or subtract predictor variables to try to achieve greater performance metric values or fewer Type 1 and Type 2 errors. We selected the list of predictor variables above as an educated guess to illustrate the algorithm.

Model 2: Artificial Neural Network

Definition

An artificial neural network (ANN) is a supervised machine learning model that is patterned after the human brain in terms of both structure and function. It consists of layers of interconnected nodes (or artificial neurons), which include an input layer that receives the data, one or more hidden layers that process the information, and an output layer that conveys the final class predictions. As with the SVM model, ANN algorithms can identify non-linear patterns in data, which makes them effective in handling complex or high-dimensional datasets.

Both SVM and ANN can be used to classify data into predefined classes, which is the goal of the examples in this white paper. The data used for ANN classification input is the same as used for the SVM.

Key Components/Steps

Below is the code used to set up and run the Random Forest model, including code comments. Readers of this white paper are free to copy the code into their own R environment and use it as written or revise it for their own needs.

Step 1: Load data

```
load("F1F2.RData")
```

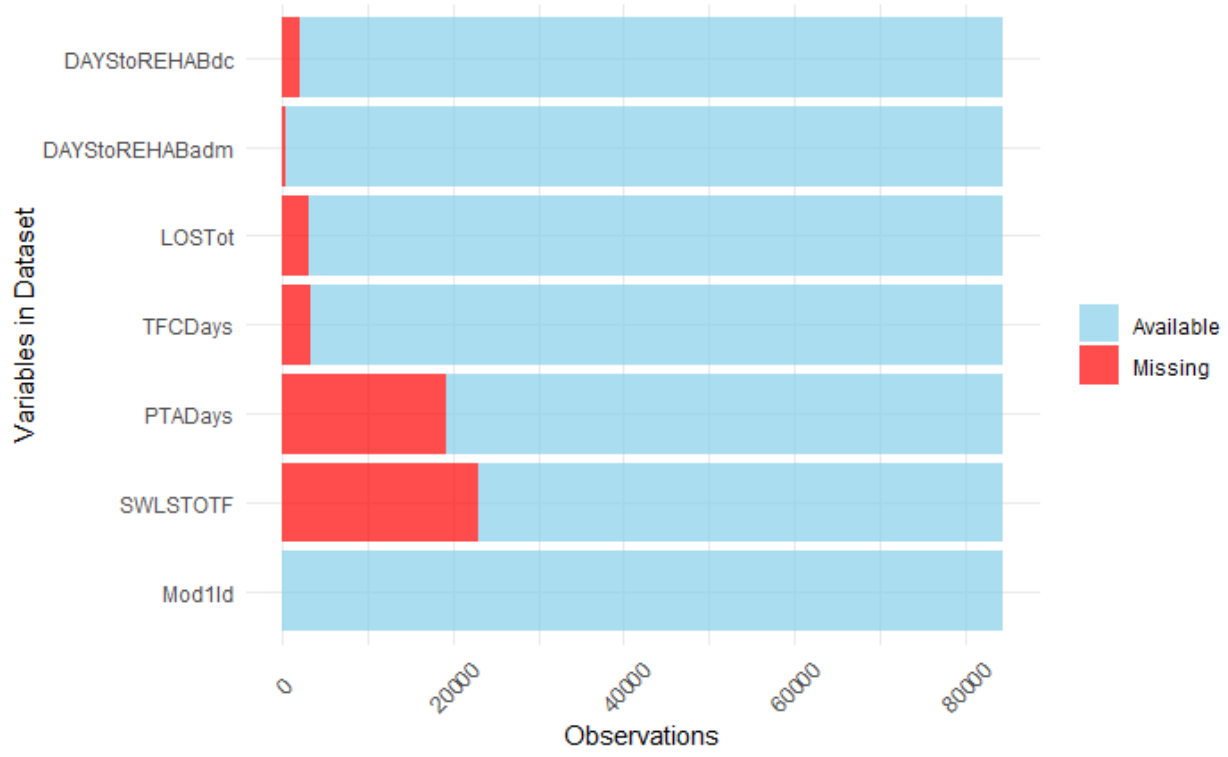
User Function to set up datasets.

```
select_columns <- function(df, vars) {
  # Check if all specified variables exist in the dataframe
  missing_vars <- setdiff(vars, names(df))

  if (length(missing_vars) > 0) {
    warning("The following variables are not in the dataframe: ", paste(missi
ng_vars, collapse = ", "))
  }

  # Select only the existing columns
  df.Selected <- df[, vars[vars %in% names(df)], drop = FALSE]

  return(df.Selected)
}

# Example usage:
#data <- data.frame(A = 1:5, B = 6:10, C = 11:15)
#vars <- c("A", "C")
#new_df <- select_columns(data, vars)
#print(new_df)
```

Step 2: Build Data for ANN

```
library(TBIMS)
library(dplyr)
library(tidyverse)

vars <- c("Mod1Id","PTADays", "TFCDays", "LOSTot", "DAYStoREHABadm", "DAYStoR
EHABdc")
df.SWLSF1 <- select_columns(df.Form1, vars)

#vars <- c("Mod1Id","SWLSTOTF", "PHQ9TOTF", "GAD7TOTF", "PARTSummaryF", "PART
OutAboutF", "PARTProductivityF", "PARTSocialF", "MarF")
vars <- c("Mod1Id","SWLSTOTF")
df.SWLSF2 <- select_columns(df.Form2, vars)

df.SWLS  <- Combine_dataframes_ID(df.SWLSF2, df.SWLSF1, "Mod1Id")
TBIMS::Missingness_Barplot(df.SWLS)
```

The bar plot output from the code shows missingness for each of the variables we selected for the model. Note that the input variables are a different subset than were selected for SVM. There are roughly 80,000 observations initially.

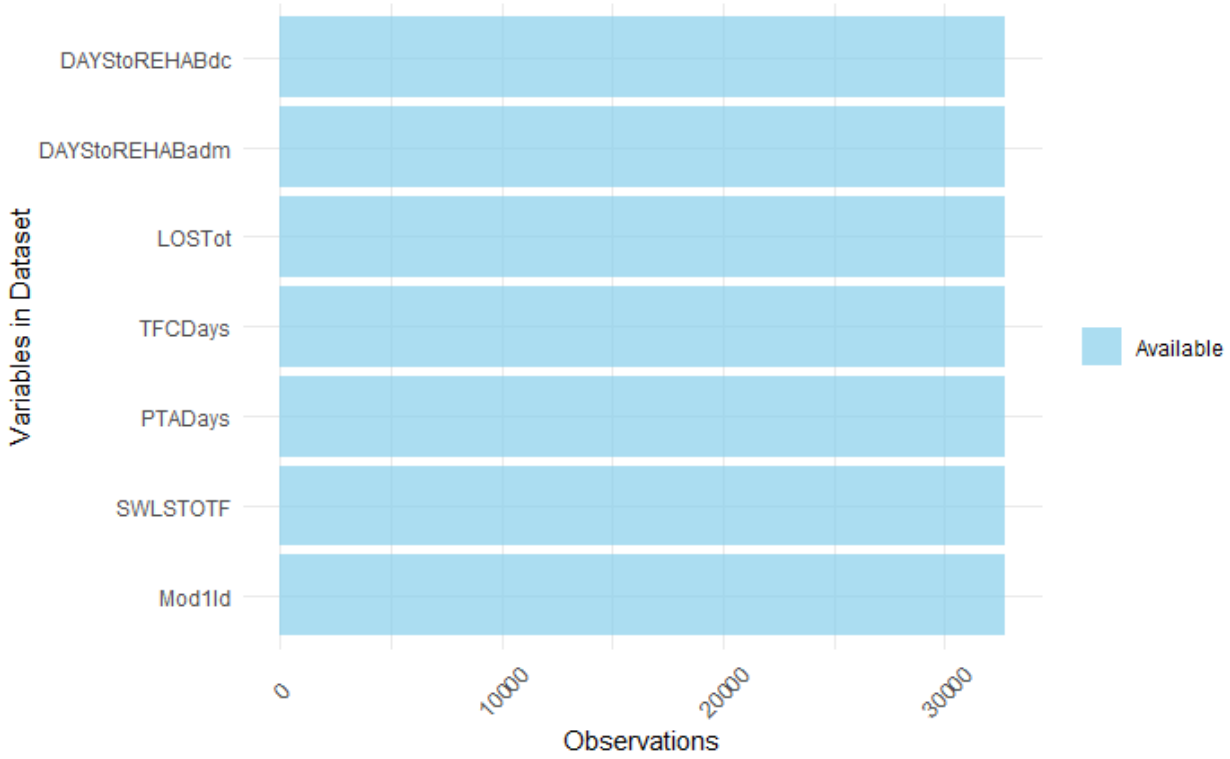


Step 3: Use Recoding Function, Address Missingness, and Create Classes

We again decided to ignore all cases with missing data and used the following code to remove them.

```
vec <- c(888, 999)
df.SWLS_1 <- TBIMS::replace_values_with_na(df.SWLS, vec)
df.SWLS_2 <- df.SWLS_1[!is.na(df.SWLS_1$SWLSTOTF),]
TBIMS::Missingness_Barplot(df.SWLS_2)
df.SWLS_3 <- df.SWLS_2[complete.cases(df.SWLS_2), ]
TBIMS::Missingness_Barplot(df.SWLS_3)
```

This reduced our number of observations to around 30,000 as can be seen in the final missingness bar plot.



Once again the variable to be classified is the Satisfaction With Life Scale (SWLS), which we divided into the two predefined classes: Extremely Dissatisfied and All Other.

```
# This function bins data from a specified column in a dataframe based on val
ue ranges provided.
#
# PARAMETERS:
#   - df: The input dataframe.
#     TYPE: dataframe
```

```
#    - column_name: The name of the column in the dataframe to bin.
#      TYPE: character
#    - value_ranges: A vector containing the value ranges to bin the data.
#      TYPE: numeric vector
#      EXAMPLE INPUT: c(0, 10, 20, 30)
# RETURNS:
#    - The input dataframe with a new column added containing the binned value
s.
#      TYPE: dataframe
# EXCEPTIONS:
#    - This function assumes the dataframe contains the specified column and t
he value ranges are in ascending order.
#    - It does not handle cases where the value to bin is outside the specifie
d ranges.
# Example Usage:
#    - Create a sample dataframe
#data <- data.frame(id = 1:5, values = c(5, 15, 25, 10, 20))
#    - Define the value ranges for binning
#ranges <- c(0, 10, 20, 30)
#    - Call the function to bin the 'values' column in the dataframe
#result_df <- bin_data(data, "values", ranges)

bin_data <- function(df, column_name, value_ranges) {
  # Create a new column to store the binned values
  df$bin_column <- cut(df[[column_name]], breaks = value_ranges, labels = FAL
SE, include.lowest = TRUE)
  return(df)
}
ranges <- c(0, 9, 35)
df.SWLS_4 <- bin_data(df.SWLS_3, "SWLSTOTF", ranges)
df.SWLS_4$bin_column <- df.SWLS_4$bin_column - 1
```

Step 4: Set Up Predictors and Target Response

We used the H2O package in R to build and train the neural network. **Disclaimer: The H2O package sends the data to an external server, runs the model on the server, and then returns the results. It is unknown what happens to the data on the server after the model is run, which is a potential security risk if the data includes any Personally Identifiable Information. The authors of this white paper have not contemplated any data security features; that is the responsibility of anyone running this code. Please check with your IT team if you have concerns or make sure not to include any sensitive data at all.**

The call to the function sets a two-minute limitation on the server connection. For this reason, the example keeps the expected processing time limited. The function accepts a dataframe, a string vector of predictors, a target label, and an integer defining the number of hidden layers between the input and output layers. For this example, the number of hidden layers is restricted to be from 1 to 5. If the user requests more than 5 hidden layers, the function warns the user that the limit is 5 and changes the number of hidden layers to 5. For each hidden layer, the square root of the number of input features should be a used for the number of neurons in

the hidden layer. The function optimizes and trains a shallow feedforward, fully connected neural network for classification. The function optimizes the following hyperparameters: learning rate, batch size, and number of epochs. The ANN uses a ReLU (Rectified Linear Unit) activation function for hidden layers and a sigmoid activation function in the output layer.

```
set.seed(2025)
# Define predictors and response
predictors <- colnames(df.SWLS_4)[3:(ncol(df.SWLS_4) - 1)]
response <- "bin_column"

train_shallow_nn_h2o <- function(data, predictors, target, num_hidden_layers,
                                 max_runtime_secs, nfolds = 5, seed = 2025) {

  # Initialize H2O
  h2o.init()
  h2o.no_progress()  # Turn off progress bars for cleaner output

  # Validate number of hidden layers
  if (num_hidden_layers > 5) {
    warning("The maximum number of hidden layers is 5. Setting
num_hidden_layers to 5.")
    num_hidden_layers <- 5
  }

  # Convert data to H2O frame and ensure target is factor for classification
  data_h2o <- as.h2o(data)
  data_h2o[[target]] <- as.factor(data_h2o[[target]])  # Force conversion to
factor

  # Define number of neurons in hidden layers (sqrt of input features)
  num_features <- length(predictors)
  hidden_units <- floor(sqrt(num_features))

  # Create hidden layer architecture vector
  hidden <- rep(hidden_units, num_hidden_layers)

  # Train classification model
  model <<- h2o.deeplearning(
    x = predictors,
    y = target,
    training_frame = data_h2o,
    hidden = hidden,
    activation = "RectifierWithDropout",  # ReLU with dropout
    standardize = TRUE,
    adaptive_rate = TRUE,                  # Automatically tunes learning rate
    nfolds = nfolds,                      # Cross-validation
    fold_assignment = "Stratified",       # For classification
    stopping_metric = "AUC",              # AUC for classification
    stopping_tolerance = 0.001,
    stopping_rounds = 5,
    max_runtime_secs = max_runtime_secs, # Limit runtime for tuning
    seed = seed,
```

```
    variable_importances = TRUE,          # Get feature importance
    export_weights_and_biases = TRUE      # Export model weights
  )

  # Get model performance metrics
  perf <- if (nfolds > 1) h2o.performance(model, xval = TRUE) else
h2o.performance(model)

  # Extract classification metrics
  metrics <<- list(
    confusion_matrix = h2o.confusionMatrix(perf),
    auc = h2o.auc(perf),
    logloss = h2o.logloss(perf),
    accuracy = h2o.accuracy(perf),
    f1 = h2o.F1(perf),
    precision = h2o.precision(perf),
    recall = h2o.recall(perf)
  )

  # Return model and metrics
  list(
    model = model,
    metrics = metrics,
    hidden_layers = hidden,
    model_summary = model@model$model_summary,
    cross_validation_metrics = if (nfolds > 1)
model@model$cross_validation_metrics_summary else NULL,
    is_classification = TRUE  # Explicitly mark as classification model
  )
}
```

Step 5: Call the ANN Function

```
library(h2o)
ANN_H2OResultList <- train_shallow_nn_h2o(
  data = df.SWLS_4,
  predictors = predictors,
  target = response,
  num_hidden_layers = 2,
  max_runtime_secs = 120
)
```

Model Fit and Results

```
Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         4 minutes 58 seconds
    H2O cluster timezone:       America/Los_Angeles
    H2O data parsing timezone:  UTC
    H2O cluster version:        3.47.0.6891
```

```
    H2O cluster version age:    5 days
    H2O cluster name:           H2O_started_from_R_emung_mld906
    H2O cluster total nodes:    1
    H2O cluster total memory:   1.43 GB
    H2O cluster total cores:    2
    H2O cluster allowed cores:  2
    H2O cluster healthy:        TRUE
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    H2O Internal Security:      FALSE
    R Version:                  R version 4.5.0 (2025-04-11 ucrt)
```

The code below is used to assess the model fit and output fit statistics and plots.

```
analyze_h2o_model <- function(h2o_model_result) {
  # Load required packages
  if (!requireNamespace("ggplot2", quietly = TRUE))
install.packages("ggplot2")
  if (!requireNamespace("gridExtra", quietly = TRUE))
install.packages("gridExtra")
  library(ggplot2)
  library(gridExtra)

  # Extract components from the model result
  model <- h2o_model_result$model
  is_classification <- h2o_model_result$is_classification
  metrics <- h2o_model_result$metrics

  # Create empty list to store plots
  plots <- list()

  # 1. Print Model Summary
  cat("=== MODEL SUMMARY ===\n")
  print(h2o_model_result$model_summary)

  # 2. Print Performance Metrics with proper null checks
  cat("\n=== PERFORMANCE METRICS ===\n")

  # Define safe_print_metric function first
  safe_print_metric <- function(metric_value, metric_name) {
    if (!is.null(metric_value)) {
      if (is.numeric(metric_value)) {
        cat(sprintf("%s: %.3f\n", metric_name, metric_value))
      } else {
        cat(sprintf("%s: %s\n", metric_name, metric_value))
      }
    } else {
      cat(sprintf("%s: Not available\n", metric_name))
    }
  }
```

```
  if (is_classification) {
    cat("Classification Model\n")
    safe_print_metric(metrics$auc, "AUC")
    safe_print_metric(metrics$logloss, "LogLoss")
    safe_print_metric(metrics$accuracy, "Accuracy")
    safe_print_metric(metrics$f1, "F1 Score")
    safe_print_metric(metrics$precision, "Precision")
    safe_print_metric(metrics$recall, "Recall")

    cat("\nConfusion Matrix:\n")
    if (!is.null(metrics$confusion_matrix)) {
      print(metrics$confusion_matrix)
    } else {
      cat("Not available\n")
    }
  } else {
    cat("Regression Model\n")
    safe_print_metric(metrics$mse, "MSE")
    safe_print_metric(metrics$rmse, "RMSE")
    safe_print_metric(metrics$mae, "MAE")
    safe_print_metric(metrics$r2, "R-squared")
  }

  # 3. Variable Importance Plot
  if (!is.null(model@model$variable_importances)) {
    varimp <- as.data.frame(h2o.varimp(model))
    if (nrow(varimp) > 0) {
      plots$varimp <- ggplot(varimp, aes(x = reorder(variable,
scaled_importance),
                                         y = scaled_importance)) +
        geom_bar(stat = "identity", fill = "steelblue") +
        coord_flip() +
        labs(title = "Variable Importance",
             x = "Features",
             y = "Scaled Importance") +
        theme_minimal()
    }
  }

  # 4. Training Metrics History Plot
  if (!is.null(model@model$training_metrics)) {
    history <-
as.data.frame(model@model$training_metrics@metrics$training_metrics)
    if (is_classification && !is.null(history$classification_error)) {
      plots$error_history <- ggplot(history, aes(x = epoch, y =
classification_error)) +
        geom_line(color = "red") +
        labs(title = "Training Error History",
             x = "Epoch",
             y = "Classification Error") +
        theme_minimal()
    }
    if (!is.null(history$mse)) {
      plots$mse_history <- ggplot(history, aes(x = epoch, y = mse)) +
```

```
        geom_line(color = "blue") +
        labs(title = "Training MSE History",
             x = "Epoch",
             y = "MSE") +
        theme_minimal()
    }
  }

  # 5. Residuals Plot (for regression)
  if (!is_classification) {
    preds <- as.data.frame(h2o.predict(model,
model@parameters$training_frame))
    actual <-
as.data.frame(model@parameters$training_frame[[model@parameters$y]])
    residuals <- actual[,1] - preds[,1]
    plots$residuals <- ggplot(data.frame(residuals = residuals), aes(x =
residuals)) +
      geom_histogram(fill = "steelblue", bins = 30) +
      labs(title = "Residuals Distribution",
           x = "Residuals",
           y = "Count") +
      theme_minimal()
  }

  # 6. ROC Curve (for classification)
  if (is_classification && !is.null(metrics$auc)) {
    perf <- h2o.performance(model)
    roc_data <- data.frame(
      fpr = perf@metrics$thresholds_and_metric_scores$fpr,
      tpr = perf@metrics$thresholds_and_metric_scores$tpr
    )
    plots$roc <- ggplot(roc_data, aes(x = fpr, y = tpr)) +
      geom_line(color = "blue") +
      geom_abline(slope = 1, intercept = 0, linetype = "dashed") +
      labs(title = sprintf("ROC Curve (AUC = %.3f)", metrics$auc),
           x = "False Positive Rate",
           y = "True Positive Rate") +
      theme_minimal()
  }

  # Display all plots
  if (length(plots) > 0) {
    cat("\n=== VISUALIZATIONS ===\n")
    grid.arrange(grobs = plots, ncol = 2)
  }

  # Return metrics and plots invisibly
  invisible(list(
    metrics = metrics,
    plots = plots
  ))
}
```

Call the analysis function.

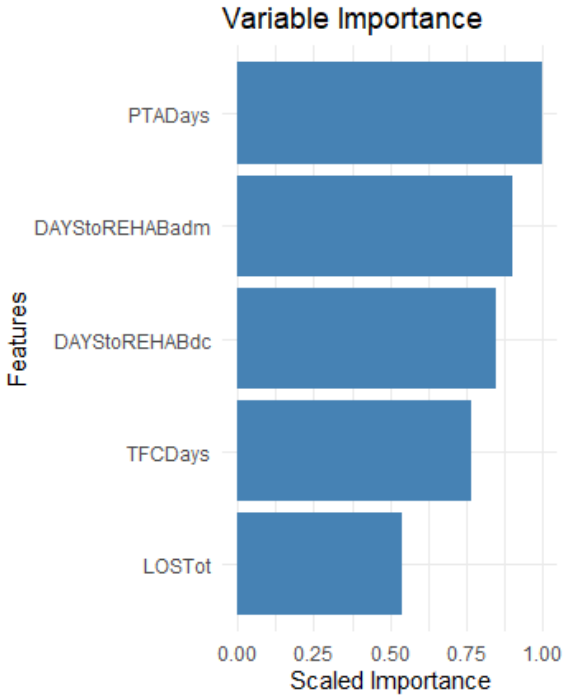
```
analysis <- analyze_h2o_model(ANN_H2OResultList)
```

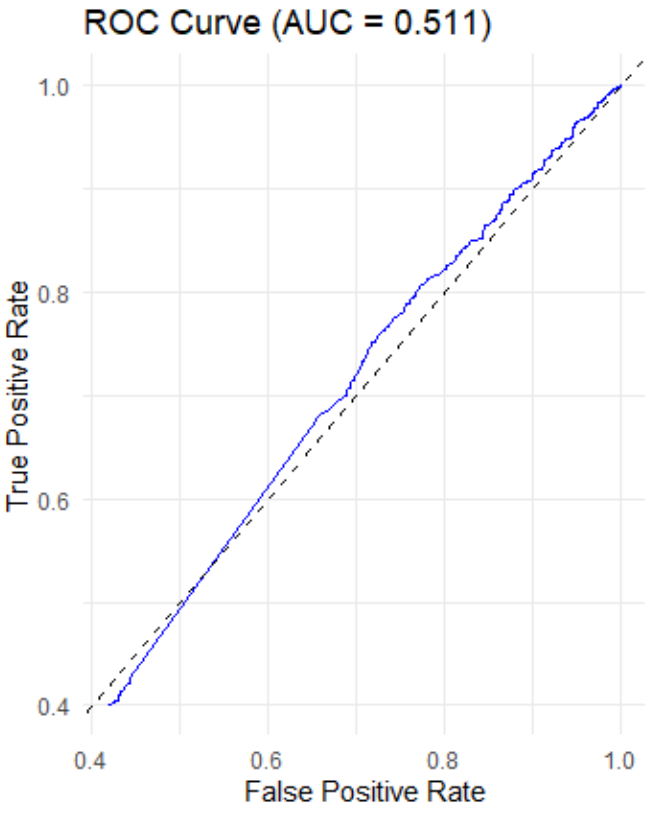
The table below shows a summary of the model, followed by a confusion matrix, a plot of variable importance, and the ROC curve.

```
=== MODEL SUMMARY ===

Status of Neuron Layers: predicting bin_column, 2-class classification, berno
ulli distribution, CrossEntropy loss, 24 weights/biases, 4.2 KB, 375,014 trai
ning samples, mini-batch size 1
```

layer	units	type	dropout	l1	l2	mean_rate	rate_rms	momentum	mean_ weight	weight_ rms	mean_ bias	bias_ rms
1	5	Input	0.00%	NA	NA	NA	NA	NA	NA	NA	NA	NA
2	2	Rectifier Dropout	50.00%	0	0	0.005409	0.002749	0	0.125718	0.461062	0.289791	0.180327
3	2	Rectifier Dropout	50.00%	0	0	0.004744	0.004863	0	-0.320067	0.493338	-0.014726	0.019906
4	2	Softmax	NA	0	0	0.007134	0.001564	0	0.635692	2.406781	0.293285	1.062551

		Actual 0	Actual 1	Error	Rate
Predicted	0	0	2957	1	1
	1	0	29800	0	0
Totals		0	32757	0.09027	0.09027


Variable Importance
PTADays
DAYStoREHABadm
DAYStoREHABdc
TFCDays
LOSTot
Features
0.00
0.25
0.50
0.75
1.00
Scaled Importance


ROC Curve (AUC = 0.511)
1.0
0.8
0.6
0.4
True Positive Rate
0.4
0.6
0.8
1.0
False Positive Rate

Model 3: Random Forest Model

Definition

Random Forest is an ensemble machine learning model that builds multiple decision trees during training and combines their predictions for improved accuracy and robustness. Unlike many other model algorithms, a Random Forest machine learning model's validation is efficiently handled through a built-in method called Out-of-Bag (OOB) Error, which eliminates the need for a separate validation set or cross-validation in many cases. This approach leverages the natural randomness of the forest's training process. Each decision tree in a Random Forest is trained on a different bootstrap sample, a random subset of the training data drawn with replacement. Due to this sampling method, approximately 60% of the data is typically used to train each tree, while the remaining ~30% (the "out-of-bag" samples) are left unseen by that particular tree.

The OOB error is calculated by using these out-of-bag samples as an implicit validation set. For every data point, only the trees that did not include it in their bootstrap training subset make predictions. These predictions are then aggregated—either by majority vote (for classification) or averaging (for regression), to generate an OOB prediction for that data point. The OOB error is then computed by comparing these predictions to the true labels, providing an unbiased estimate of the model's generalization performance, much like cross-validation.

One of the key advantages of OOB error is its efficiency, as it allows model validation without sacrificing additional data for a hold-out set or requiring computationally expensive cross-validation. It also helps in tuning hyperparameters, such as the number of trees or their depth, by monitoring how the OOB error changes during training. Overall, the OOB error serves as a robust and convenient validation mechanism intrinsic to Random Forests, ensuring reliable performance assessment while optimizing the training process.

Key Components/Steps

Below is the code used to set up and run the Random Forest model, including code comments. Readers of this white paper are free to copy the code into their own R environment and use it as written or revise it for their own needs.

Step 1: Load data

```
load("F1F2.RData")
```

Note that this is the same dataset that was used in the SVM model.

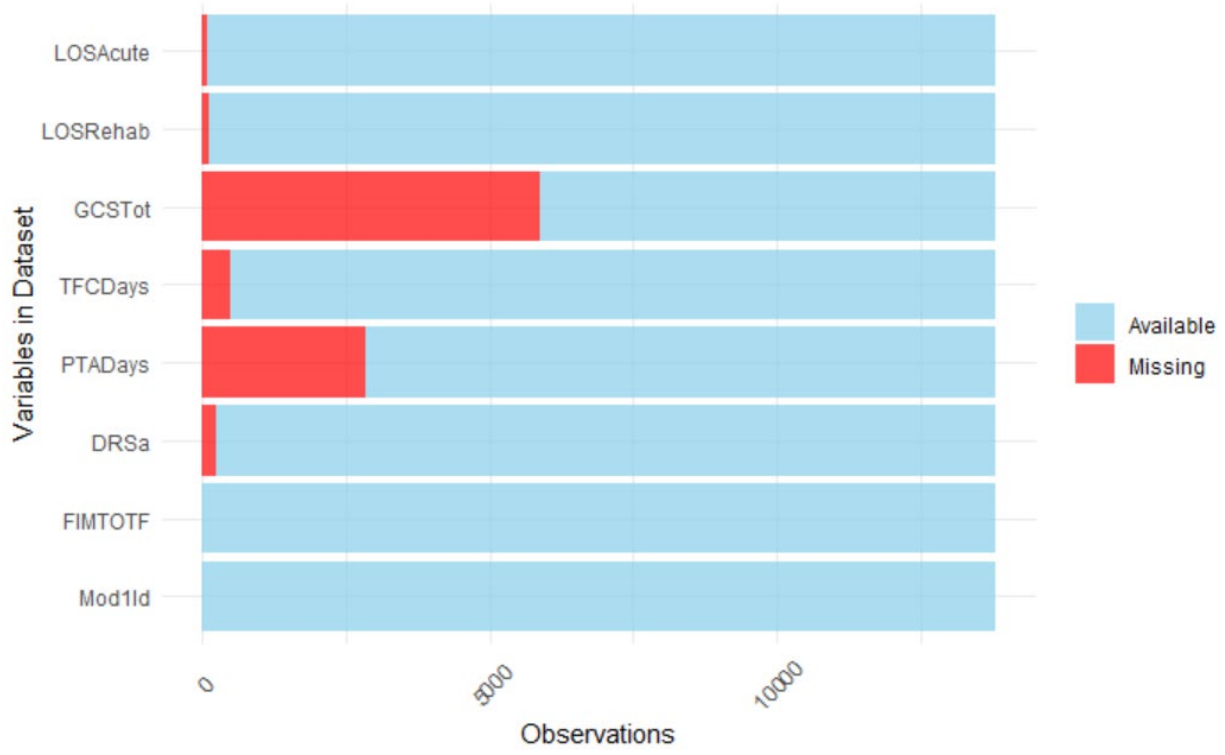
```
select_columns <- function(df, vars) {
  # Check if all specified variables exist in the dataframe
  missing_vars <- setdiff(vars, names(df))

  if (length(missing_vars) > 0) {
```

```
    warning("The following variables are not in the dataframe: ", paste(missi
ng_vars, collapse = ", "))
  }

  # Select only the existing columns
  df.Selected <- df[, vars[vars %in% names(df)], drop = FALSE]

  return(df.Selected)
}

# Example usage:
#data <- data.frame(A = 1:5, B = 6:10, C = 11:15)
#vars <- c("A", "C")
#new_df <- select_columns(data, vars)
#print(new_df)
```

Step 2: Build data for training and testing of the Random Forest model

```
library(TBIMS)
library(dplyr)
library(tidyverse)

vars <- c("Mod1Id","DRSa","PTADays", "TFCDays", "GCSTot", "LOSRehab", "LOSAcu
te")
df.RF1 <- select_columns(df.Form1, vars)

df.Form2_2 <- df.Form2[df.Form2[["FollowUpPeriod"]] %in% 2, ]

vars <- c("Mod1Id","FIMTOTF")
df.RF2 <- select_columns(df.Form2_2, vars)

df.RF3  <- Combine_dataframes_ID(df.RF2, df.RF1, "Mod1Id")

# Some values need to be removed from the training set because the taget is N
A.
values <- c(9999, NA)
df.RF <- df.RF3[!df.RF3[["FIMTOTF"]] %in% values, ]

TBIMS::Missingness_Barplot(df.RF)
```

The bar plot output from the code shows missingness for each of the variables we selected for the model. Note that the Random Forest algorithm can handle missing data.



Step 3: Run the user function to train the Random Forest model, and output training and test data

This R function performs Bayesian Optimization to tune a Random Forest regression model with the specifications provided by the user. It uses the rBayesianOptimization package along with randomForest for training. The function handles missing values by imputing them multiple ways (e.g., mean and mode), training multiple trees, and averaging predictions, approximating the idea of soft routing. This approach mimics ignoring missing values during the split point calculation and pushing them down both paths of the split with reduced weight. For true dual-path missing handling, we would need to implement a custom tree or use XGBoost, which approximates that logic (sends missing values to the direction with the best gain).

Basic features of this function include:

- No one-hot encoding needed — factors handled directly
- Simulates dual-path splitting for missing values by duplicating imputed rows
- Uses ranger for speed and flexibility
- Bayesian optimization over 5 hyperparameters
- Returns the trained model

```
optimize_rf_model <- function(df, predictors, target, seed = 2025) {
  library(rBayesianOptimization)
  library(caret)
```

```
  library(dplyr)
  library(ranger)

  set.seed(seed)

  # Subset data
  data <- df[, c(predictors, target)]

  # Ensure categorical variables are factors
  for (col in predictors) {
    if (is.character(data[[col]])) {
      data[[col]] <- as.factor(data[[col]])
    }
  }

  # Split into train/test
  train_idx <- createDataPartition(data[[target]], p = 0.8, list = FALSE)
  train_data <<- data[train_idx, ]
  test_data <<- data[-train_idx, ]

  # Function to handle NA by duplicating rows with mean/mode imputations
  dual_impute <- function(df) {
    num_vars <- names(df)[sapply(df, is.numeric)]
    cat_vars <- names(df)[sapply(df, is.factor)]

    impute_mean <- df
    impute_mode <- df

    for (var in num_vars) {
      impute_mean[[var]][is.na(impute_mean[[var]])] <-
mean(impute_mean[[var]], na.rm = TRUE)
      impute_mode[[var]][is.na(impute_mode[[var]])] <-
as.numeric(names(which.max(table(impute_mode[[var]]))))
    }

    for (var in cat_vars) {
      mode_val <- names(which.max(table(impute_mode[[var]])))
      impute_mean[[var]][is.na(impute_mean[[var]])] <- mode_val
      impute_mode[[var]][is.na(impute_mode[[var]])] <- mode_val
    }

    rbind(impute_mean, impute_mode)
  }

  # Optimization target function
  rf_cv <- function(num.trees, max.depth, mtry, min.node.size,
sample.fraction) {
    dtrain <- dual_impute(train_data)
    dtest <- dual_impute(test_data)

    model <- ranger(
      formula = as.formula(paste(target, "~ .")),
      data = dtrain,
      num.trees = round(num.trees),
```

```
      mtry = max(1, floor(mtry)),
      max.depth = round(max.depth),
      min.node.size = round(min.node.size),
      sample.fraction = min(sample.fraction, 1),
      seed = seed,
      classification = FALSE
    )

    preds <- predict(model, data = dtest)$predictions
    rmse <- sqrt(mean((preds - dtest[[target]])^2))

    list(Score = -rmse)
  }

  bounds <- list(
    num.trees = c(100L, 500L),
    max.depth = c(3L, 30L),
    mtry = c(1L, length(predictors)),
    min.node.size = c(1L, 10L),
    sample.fraction = c(0.5, 1)
  )

  opt_result <- BayesianOptimization(
  FUN = rf_cv,
  bounds = bounds,
  init_points = 8,
  n_iter = 15,
  acq = "ei",
  verbose = TRUE
  )

  # Train final model with best params
  full_data <- dual_impute(data)
  best <- opt_result$Best_Par

  final_model <- ranger(
    formula = as.formula(paste(target, "~ .")),
    data = full_data,
    num.trees = round(best[["num.trees"]]),
    mtry = max(1, floor(best[["mtry"]])),
    max.depth = round(best[["max.depth"]]),
    min.node.size = round(best[["min.node.size"]]),
    sample.fraction = min(best[["sample.fraction"]], 1),
    seed = seed,
    classification = FALSE,
    importance = "impurity"
  )

  return(final_model)
}
```

Step 4: Call the random forest training function

For purposes of our example, we selected the FIM Total Score at Follow-up (FIMTOTF) to be the outcome/dependent variable and a small set of predictors to limit the runtime of the model.

```
predictors = c(vars <- c("DRSa","PTADays", "TFCDays", "GCSTot", "LOSRehab", "
LOSAcute"))
target = "FIMTOTF"

model <- optimize_rf_model(
  df = df.RF,
  predictors = predictors,
  target = target
)
```

The training function produced the following results.

```
elapsed = 5.97  Round = 1   num.trees = 215.0000    max.depth = 27.0000 mtry
= 2.0000   min.node.size = 8.0000  sample.fraction = 0.5083176 Value = -17.07
367
elapsed = 1.50  Round = 2   num.trees = 394.0000    max.depth = 4.0000  mtry
= 1.0000   min.node.size = 10.0000 sample.fraction = 0.9911107 Value = -17.79
307
elapsed = 1.61  Round = 3   num.trees = 144.0000    max.depth = 12.0000 mtry
= 2.0000   min.node.size = 1.0000  sample.fraction = 0.5523794 Value = -17.00
951
elapsed = 3.89  Round = 4   num.trees = 262.0000    max.depth = 25.0000 mtry
= 2.0000   min.node.size = 9.0000  sample.fraction = 0.5267616 Value = -17.04
502
elapsed = 6.82  Round = 5   num.trees = 351.0000    max.depth = 22.0000 mtry
= 2.0000   min.node.size = 6.0000  sample.fraction = 0.8584217 Value = -17.21
421
elapsed = 1.19  Round = 6   num.trees = 265.0000    max.depth = 4.0000  mtry
= 2.0000   min.node.size = 4.0000  sample.fraction = 0.7746331 Value = -17.49
827
elapsed = 1.77  Round = 7   num.trees = 264.0000    max.depth = 8.0000  mtry
= 2.0000   min.node.size = 4.0000  sample.fraction = 0.588202  Value = -17.01
472
elapsed = 6.36  Round = 8   num.trees = 467.0000    max.depth = 15.0000 mtry
= 2.0000   min.node.size = 10.0000 sample.fraction = 0.9393052 Value = -17.08
059
elapsed = 7.97  Round = 9   num.trees = 486.0000    max.depth = 28.0000 mtry
= 2.0000   min.node.size = 2.0000  sample.fraction = 0.5260243 Value = -17.21
573
elapsed = 0.91  Round = 10  num.trees = 149.0000    max.depth = 9.0000  mtry
= 2.0000   min.node.size = 3.0000  sample.fraction = 0.5232026 Value = -17.03
313
elapsed = 2.78  Round = 11  num.trees = 180.0000    max.depth = 27.0000 mtry
= 2.0000   min.node.size = 2.0000  sample.fraction = 0.5565545 Value = -17.22
608
```

```
elapsed = 1.78  Round = 12  num.trees = 394.0000    max.depth = 6.0000  mtry
= 2.0000   min.node.size = 7.0000  sample.fraction = 0.9982801 Value = -17.19
377
elapsed = 4.38  Round = 13  num.trees = 290.0000    max.depth = 28.0000 mtry
= 2.0000   min.node.size = 2.0000  sample.fraction = 0.5276458 Value = -17.15
635
elapsed = 1.36  Round = 14  num.trees = 218.0000    max.depth = 11.0000 mtry
= 2.0000   min.node.size = 9.0000  sample.fraction = 0.5462984 Value = -16.98
455
elapsed = 2.53  Round = 15  num.trees = 300.0000    max.depth = 17.0000 mtry
= 2.0000   min.node.size = 9.0000  sample.fraction = 0.5134841 Value = -17.01
019
elapsed = 6.06  Round = 16  num.trees = 370.0000    max.depth = 23.0000 mtry
= 3.0000   min.node.size = 8.0000  sample.fraction = 0.8471415 Value = -17.38
27
elapsed = 4.07  Round = 17  num.trees = 260.0000    max.depth = 28.0000 mtry
= 2.0000   min.node.size = 7.0000  sample.fraction = 0.8902364 Value = -17.26
602
elapsed = 1.90  Round = 18  num.trees = 240.0000    max.depth = 14.0000 mtry
= 2.0000   min.node.size = 7.0000  sample.fraction = 0.5468118 Value = -16.99
366
elapsed = 1.51  Round = 19  num.trees = 256.0000    max.depth = 11.0000 mtry
= 2.0000   min.node.size = 7.0000  sample.fraction = 0.5578687 Value = -16.97
165
elapsed = 2.83  Round = 20  num.trees = 162.0000    max.depth = 17.0000 mtry
= 5.0000   min.node.size = 2.0000  sample.fraction = 0.6589699 Value = -17.59
996
elapsed = 1.61  Round = 21  num.trees = 164.0000    max.depth = 13.0000 mtry
= 2.0000   min.node.size = 9.0000  sample.fraction = 0.9731164 Value = -17.06
601
elapsed = 2.03  Round = 22  num.trees = 494.0000    max.depth = 7.0000  mtry
= 2.0000   min.node.size = 4.0000  sample.fraction = 0.6256199 Value = -17.06
467
elapsed = 2.68  Round = 23  num.trees = 454.0000    max.depth = 10.0000 mtry
= 2.0000   min.node.size = 7.0000  sample.fraction = 0.5653047 Value = -16.99
275

 Best Parameters Found:
Round = 19  num.trees = 256.0000    max.depth = 11.0000 mtry = 2.0000   min.n
ode.size = 7.0000  sample.fraction = 0.5578687 Value = -16.97165
```

The model returned is an optimized **Ranger Random Forest** model. Several key hyperparameters control the model's training process, influencing its performance, computational efficiency, and generalization ability. The **number of trees (num.trees)** determines how many individual decision trees are grown in the forest. A larger number generally improves prediction stability and reduces overfitting, but with diminishing returns and increased computational cost. The **maximum depth of trees (max.depth)** limits how many splits a tree can have, controlling complexity: deeper trees can capture finer patterns but risk overfitting, while shallower trees promote generalization at the cost of underfitting.

The **mtry** hyperparameter defines the number of randomly selected features considered for splitting at each node. A smaller mtry increases diversity among trees (helpful for high-dimensional data), while a larger value makes trees more similar but potentially more accurate when features are highly informative. The **minimum node size (min.node.size)** sets the smallest number of observations allowed in a terminal node, influencing tree granularity, smaller nodes capture more detail but may overfit, whereas larger nodes create smoother, more generalized predictions.

Finally, the **sample fraction (sample.fraction)** determines the proportion of training data randomly sampled (with replacement) for each tree. A lower fraction increases randomness and speeds up training, while a value of 1.0 (using all data) may reduce variance but can make trees more correlated.

```
Ranger result

Call:
 ranger(formula = as.formula(paste(target, "~ .")), data = full_data,      nu
m.trees = round(best[["num.trees"]]), mtry = max(1, floor(best[["mtry"]])),
max.depth = round(best[["max.depth"]]), min.node.size = round(best[["min.node
.size"]]),      sample.fraction = min(best[["sample.fraction"]], 1), seed = s
eed,      classification = FALSE, importance = "impurity")

Type:                             Regression
Number of trees:                  256
Sample size:                      27626
Number of independent variables:  6
Mtry:                             2
Target node size:                 7
Variable importance mode:         impurity
Splitrule:                        variance
OOB prediction error (MSE):       239.188
R squared (OOB):                  0.3457674
```

Note that for the Variable Importance Mode we chose the Impurity algorithm in this training set. There are other options that users may choose, including Mean Squared Error, permutation importance or Lasso regression.

Step 5: Use the test data to get the true error of the model and rank the importance of the top 10 variables

The model fit data above is based on the OOB error. We divided the training data for this model into 80% training and 20%. In machine learning, the **training data** and **testing data** serve distinct but complementary purposes in developing and evaluating a predictive model. The **training data** is the subset of the dataset used to teach the model by adjusting its parameters, such as split criteria in a decision tree, to minimize errors in prediction. This phase involves learning patterns, relationships, and structures within the data. However, evaluating a model

solely on the training data can be misleading because the model may **overfit**, meaning it performs exceptionally well on the training examples but fails to generalize to unseen data.

To assess the model's true performance and generalizability, the **testing data**, our separate 20% of the dataset held-out and not used during training, is employed. This data acts as an unbiased benchmark, simulating real-world scenarios where the model encounters new, previously unseen inputs. By comparing the model's predictions on the testing data against the actual outcomes, we can measure its accuracy, robustness, and ability to generalize beyond memorized training examples. Metrics such as classification accuracy, mean squared error, or precision-recall scores are computed on the testing data to gauge model fit. If performance is significantly worse on the testing set than the training set, it indicates overfitting, prompting strategies such as regularization, cross-validation, or model simplification.

This is the code for the user function for testing the model:

```
analyze_ranger_model <- function(model, data) {
  library(ggplot2)
  library(Metrics)
  library(dplyr)

  # Predict on provided data
  predictions <- predict(model, data = data)$predictions
  actuals <- data[[model$dependent.variable.name]]

  # Calculate fit statistics
  rmse_val <- rmse(actuals, predictions)
  r2_val <- 1 - sum((actuals - predictions)^2) / sum((actuals -
mean(actuals))^2)
  mae_val <- mae(actuals, predictions)

  fit_stats <- data.frame(
    RMSE = rmse_val,
    R2 = r2_val,
    MAE = mae_val
  )

  print("Fit Statistics:")
  print(fit_stats)

  # Plot 1: Actual vs Predicted
  p1 <- ggplot(data.frame(Actual = actuals, Predicted = predictions), aes(x =
Actual, y = Predicted)) +
    geom_point(alpha = 0.6, color = "#0072B2") +
    geom_abline(slope = 1, intercept = 0, linetype = "dashed", color =
"gray40") +
    labs(title = "Predicted vs Actual", x = "Actual", y = "Predicted") +
    theme_minimal()

  print(p1)
```

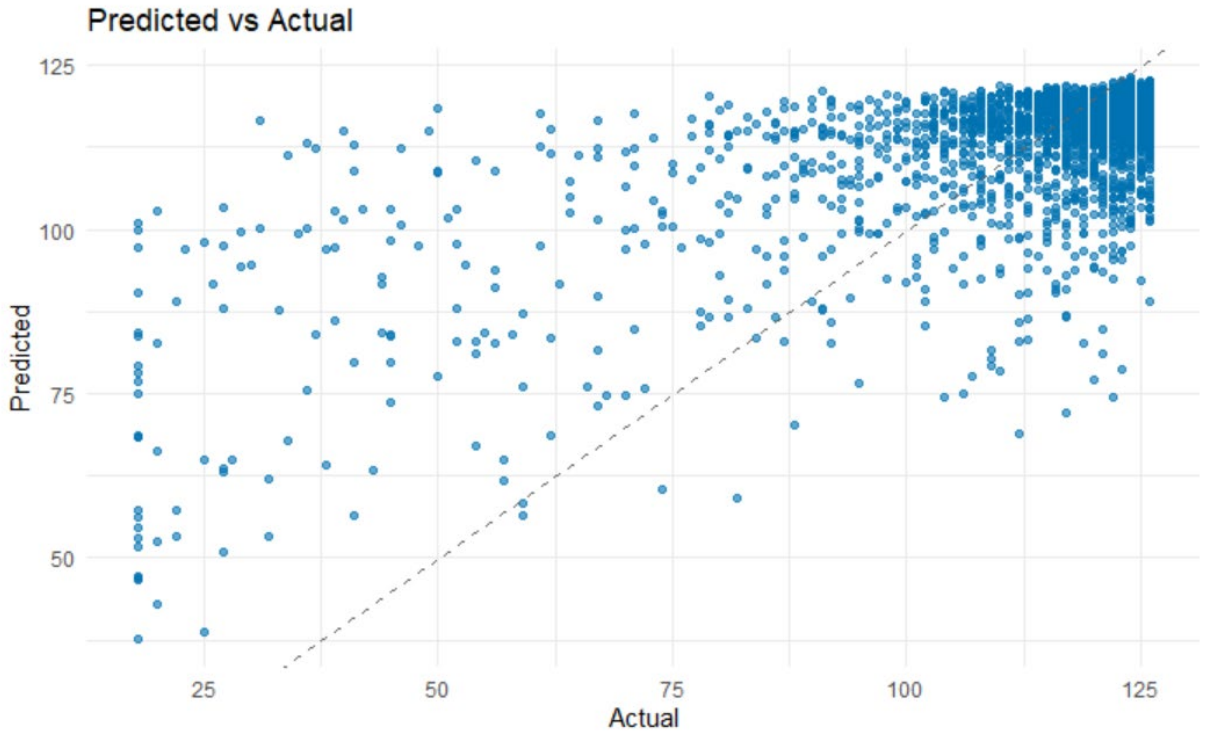
```
  # Plot 2: Top 10 Variable Importance
  if (!is.null(model$variable.importance)) {
    importance_df <- data.frame(
      Variable = names(model$variable.importance),
      Importance = model$variable.importance
    ) %>%
      arrange(desc(Importance)) %>%
      slice_head(n = 10)

    p2 <- ggplot(importance_df, aes(x = reorder(Variable, Importance), y =
Importance)) +
      geom_col(fill = "#D55E00") +
      coord_flip() +
      labs(title = "Top 10 Important Variables", x = "", y = "Importance") +
      theme_minimal()

    print(p2)
  } else {
    message("Variable importance not available in model.")
  }

  return(invisible(fit_stats))
}
```

Step 6: Analyze the fit of the testing function

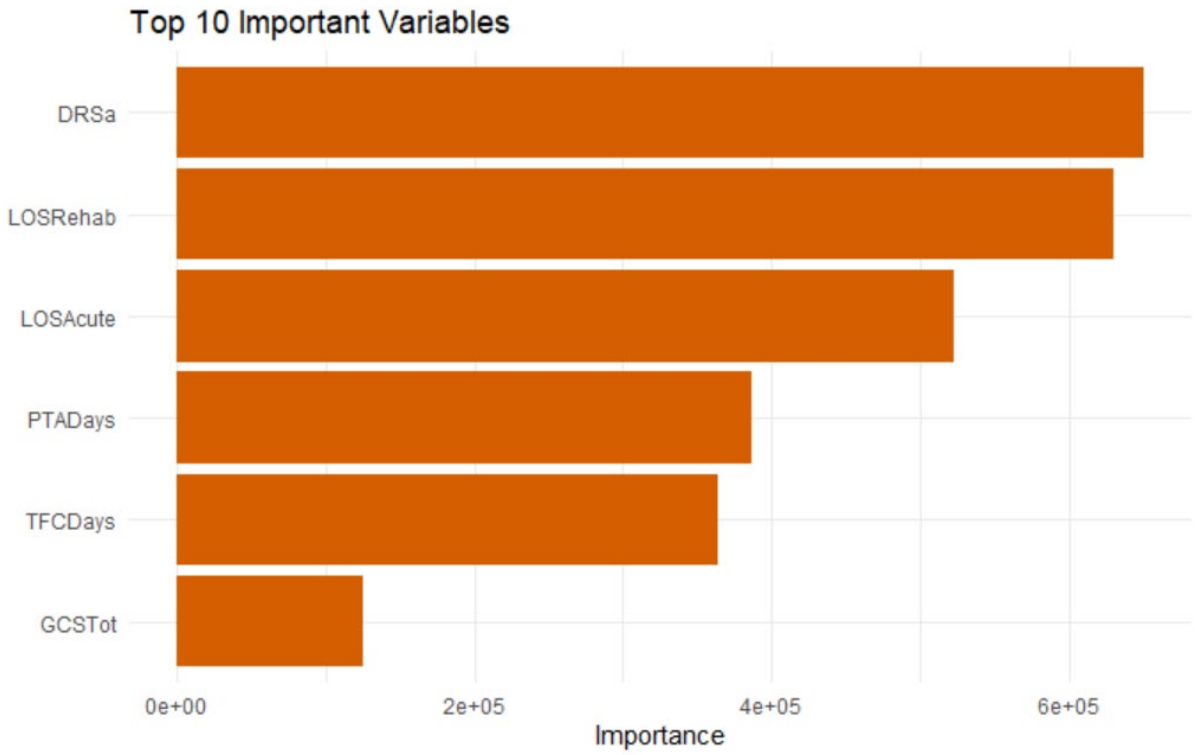
```
analyze_ranger_model(model = model, data = test_data)
[1] "Fit Statistics:"
```

RMSE	R2	MAE
<dbl>	<dbl>	<dbl>
14.74421	0.4413151	9.466461

As with the SVM model, we generated a plot of predicted versus true values using the test data. In the figure below, Actual FIM Total scores at Follow-up are on the x-axis and Predicted scores are on the y-axis. Notice that the model predicts high scores with great accuracy but does not predict low scores well. Thus, a significant portion of the error comes from the model's inability to predict lower values.



Finally, the model returned the top ten variables of importance. Note that, while the algorithm looked for ten variables, it only found six. This is the result of our minimal set of predictor variables for the sake of reducing model runtime. If users select a larger set of predictors, the model will return the top ten of importance.



Appendix

TBIMS Missing Data Codes

The table below contains variables with special 8-based missing codes. These codes exist for these variables in addition to any 8, 88, 888, or 8888 codes that reflect a value of Not Applicable. The final column in the table shows how we defined data availability for a particular code.

Variable Group	Item	Code	Form 1 or 2	Availability Classification
GAD	If the interview is being done by proxy (vs the participant themselves), all items are coded **82 - Not Applicable: No data from person with TBI**	82	2	Not Available
	If first two items (GADNervousF and GADCntrlWryF) are coded 0 - Not at All, then the remaining items are coded **81 - Not Applicable** Remaining items are GADWorryF, GADRelaxF, GADRestlessF, GADAnnoyF, GADAfraidF, GADDifficultF	81	2	Not Available
CARE Tool	This is for all variables that begin with MOB, for Mobility, or SC, for Self-Care. If activity was not attempted, code the reason:		1	
	81 - Not applicable - Not attempted and the patient/resident did not perform this activity prior to the current illness, exacerbation, or injury.	81		
	82 - Not attempted due to environmental limitations (e.g., lack of equipment, weather constraints)	82		
	83 - Not attempted due to medical condition or safety concerns	83		

Variable Group	Item	Code	Form 1 or 2	Availability Classification
	84 - Did Not Meet Criteria for Administration (To be used if participant leaves AMA, returns to ICU and does not return to rehab, or is only on rehab unit for 24 hours or less).	84		
IntStatus	**87 - Future FollowUpPeriod**	87	2	
Collection Methods	Includes the following variables:		2	
	CollectionMethodPrimaryF **81 - NA: Funding Not Available**	81		Not Available
	CollectionMethodPrimaryF **82 - Not Applicable**	82		Not Available
	CollectionMethodSecondaryF **81 - NA: Funding Not Available**	81		Not Available
	CollectionMethodSecondaryF **82 - NA: No Secondary Method of Data Collection**	82		Not Available
	LostReasonF **81 - Not Applicable**	81		1 - Not Lost
	LostReasonF **82 - Not Applicable, Expired**	82		1 - Not Lost
	LostReasonF **83 - Not Applicable, Funding Not Available**	83		0 - Lost
	ReasonNoDataIndF **81 - Not Applicable, Funding Not Available**	81		0 - Lost
	ReasonNoDataIndF **82 - Not Applicable, Data Was Provided**	82		1 - Not Lost
	LengthInterviewF **8881 - NA-Data Collected Online**	8881		Not Available
	LengthInterviewF **8882 - NA-Data Collected by Mail-Out**	8882		Not Available
PHQ	If the interview is being done by proxy (vs the participant themselves), all items are coded **82 - Not Applicable: No data from person with TBI**	82	2	Not Available

Variable Group	Item	Code	Form 1 or 2	Availability Classification
	If first two items (PHQPleasureF and PHQDownF) are coded 0 - Not at All, then the remaining items are coded **81 - Not Applicable** Remaining items are PHQSleepF, PHQTiredF, PHQEatF, PHQBadF, PHQConcentrateF, PHQSlowF, PHQDeadF, PHQDifficultF	81	2	Not Available
Employment	Includes the following variables:		2	
	DaysTo1stEmpF: **88888 - No Competitive Employment Since Injury**	88888		
	DaysTo1stEmpF: **88999 - Began Competitive Employment in Prior Follow-Up Year**	88899		Could pull the value from the previous follow-up interview.
Alcohol	Includes the following variables:		Both 1 and 2	
	ALC4DrinksF, ALC5DrinksF, ALCDrinksF: **881 - Not Applicable**	881		Not Available
	ALC4DrinksF, ALC5DrinksF, ALCDrinksF: **882 - Not Applicable: Variable not due this year (Code no longer used; data now collected in all follow-up years)**	882		Not Available
	ALCAnyDrinkF: **81 - Not Applicable**	81		Not Available
	ALCAnyDrinkF: **82 - Not Applicable: Variable not due this year (Code no longer used; data now collected in all follow-up years)**	82		Not Available
	ALCWeekF: **81 - Not Applicable**	81		Not Available
	ALCWeekF: **82 - Not Applicable: Variable not due this year**	82		Not Available

Variable Group	Item	Code	Form 1 or 2	Availability Classification
SWLS	Includes the following variables:		2	
	SWLSCondF, SWLSIdealF, SWLSImprtF, SWLSSAtF: **81 - Not Applicable: Variable not due this year (Code no longer used; data now collected in all follow-up years)**	81		Not Available
	SWLSCondF, SWLSIdealF, SWLSImprtF, SWLSSAtF: **82 - Not Applicable: No data from person with TBI**	82		Not Available
Transportation	Includes the following variables:		2	
	TransModeF: **81 - Not Applicable: Variable not due this year (Code no longer used; data now collected in all follow-up years)**	81		Not Available
	TransModeF: **82 - Not Applicable: No motorized transportation**	82		Could do 0 = No Motorized Transportation OR 5 = Non-Motorized Transportation

Data Transformations

The table below shows which variables are defined as continuous, which are categorical whose categories could be collapsed if desired, which benefit from being dummy encoded, and variables which change over time (such as having separate variables for admission and discharge).

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
Data						
1	SubjectId					
1	Birth					
1	Center	TBIMS Centers		X	X	
1	DataFrom	9 Categories		X	X	
1	DataMethod					

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	Death					
1	DeathCause1					
1	DeathCause2					
1	DeathECode					
Demographic						
1	Age	Calculated	X			
1	BMI	Calculated	X			
1	BMICat	8 Categories		X		
1	Height	Used for BMI calculation	X			
1	Weight	Used for BMI calculation	X			
1	SexF	Binary				
1	LngSpkHmF	3 Categories		X	X	
1	LngSpkHmOthF	Free form answers based on LngSpkHmF		X		
1	EthnicityF	Binary				
1	RaceAsnF	Binary				
1	RaceBlkF	Binary				
1	RaceIndF	Binary				
1	RacePIF	Binary				
1	RaceWhtF	Binary				
1	EMPLOYMENT	10 Categories (Calculated)		X		
1	Emp1	15 Categories		X		
1	Earn	11 Categories (large missingness)		X		
1	OCC	14 Categories		X		
1	EDUCATION	12 Categories (Calculated)		X		
1	EduYears	21 Categories		X		
1	GED	Binary				

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	SpEd	Binary				
1	LivWhoDis	4 Categories		X	X	Discharge
1	LivWhoInj	4 Categories		X	X	Injury
1	ResDis	10 Categories		X	X	Discharge
1	ResInj	10 Categories		X	X	Injury
1	Mar	6 Categories		X		
1	MILCombatF	Binary				
1	MILYearsF					
1	MntlEver	Binary				
1	MntlPrior	Binary				
1	PsyHosp	Binary				
1	PsyHospPrior	Binary				
1	Suicide	Binary				
1	SuicidePrior	Binary				
1	ZipDis					Discharge
1	ZipInj					Injury
Drugs/Alcohol						
1	ALC4Drinks	Count	X			
1	ALC5Drinks	Count	X			
1	ALCAnyDrink	Binary				
1	ALCDrinks	Count	X			
1	ALCWeek	Count	X			
1	DRINKCat	4 Categories (Calculated)		X		
1	PROBLEMUse	Binary (Calculated)				
1	Drugs	Binary				
1	MJPrescribe	Binary				
1	MJUse	Binary				
1	SmkCig	3 Categories		X	X	
Injury						
1	AcutePay1	10 Categories		X	X	
1	AcutePay2	10 Categories		X	X	

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	RehabPay1	10 Categories		X	X	
1	RehabPay2	10 Categories		X	X	
1	Cause	18 Categories		X		
1	CauseE1	ICD Codes				
1	CauseE2	ICD Codes				
1	SCI	Binary				
1	Craniotomy	4 Categories		X	X	
1	HospSeiz	Binary				
1	Seiz24	Binary				
1	Seiz24to7	Binary				
1	Seiz7Plus	Binary				
1	LOSAcute	Calculated	X			
1	LOSRehab	Calculated	X			
1	LOA1End					
1	LOA1Start					
1	LOA2End					
1	LOA2Start					
OSU-TBI						
1	cntAnyInjuries	Count	X	X	X	
1	cntAnyAfterIndex	Count	X	X	X	
1	cntAnyBeforeIndex	Count	X	X	X	
1	cntAnyBefore15yr	Count	X	X	X	
1	cntAnySameIndex	Count	X	X	X	
1	cntLOCInjuries	Count	X	X	X	
1	cntLOCAfterIndex	Count	X	X	X	
1	cntLOCBeforeIndex	Count	X	X	X	
1	cntLOCBefore15yr	Count	X	X	X	
1	cntLOCSameIndex	Count	X	X	X	
1	cntModSevInjuries	Count	X	X	X	
1	cntModSevAfterIndex	Count	X	X	X	
1	cntModSevBeforeIndex	Count	X	X	X	
1	cntModSevBefore15yr	Count	X	X	X	
1	cntModSevSameIndex	Count	X	X	X	

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	MostSevere	Count	X	X	X	
1	TBI_IDAsked					
1	YoungestAgeTBI	Continuous	X			
CT						
1	CTStatus	CT variables dependent on this question				
1	CT5a1CorticalLFront	Binary				
1	CT5a2CorticalRFront	Binary				
1	CT5a3CorticalNFront	Binary				
1	CT5b1CorticalLTemp	Binary				
1	CT5b2CorticalRTemp	Binary				
1	CT5b3CorticalNTemp	Binary				
1	CT5c1CorticalLPar	Binary				
1	CT5c2CorticalRPar	Binary				
1	CT5c3CorticalNPar	Binary				
1	CT5d1CorticalLOcc	Binary				
1	CT5d2CorticalROcc	Binary				
1	CT5d3CorticalNOcc	Binary				
1	CT5e1CorticalLUnk	Binary				
1	CT5e2CorticalRUnk	Binary				
1	CT5e3CorticalNUnk	Binary				
1	CT6aNonCortL	Binary				
1	CT6aNonCortN	Binary				
1	CT6aNonCortR	Binary				
1	CT7a1AxialLEpi	Binary				
1	CT7a2AxialREpi	Binary				
1	CT7a3AxialNEpi	Binary				
1	CT7b1AxialLSub	Binary				
1	CT7b2AxialRSub	Binary				
1	CT7b3AxialNSub	Binary				
1	CT7c1AxialLNS	Binary				
1	CT7c2AxialRNS	Binary				

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	CT7c3AxialNNS	Binary				
1	CT7d1FalcineSub	Binary				
1	CT7d2FalcineSAH	Binary				
1	CT7d3FalcineUnk	Binary				
1	CTComp	5 Categories		X		
1	CTFrag	Binary				
1	CTIntracrain	Binary				
1	CTIntraventricular	Binary				
1	CTPunctate	Binary				
1	CTSubarachnioid	Binary				
DRS						
1	DRSA	Calculated	X			Admission
1	DRSD	Calculated	X			Discharge
1	DRSEmpA		X			Admission
1	DRSEmpD		X			Discharge
1	DRSEyeA		X			Admission
1	DRSEyeD		X			Discharge
1	DRSFeedA		X			Admission
1	DRSFeedD		X			Discharge
1	DRSFuncA		X			Admission
1	DRSFuncD		X			Discharge
1	DRSGroomA		X			Admission
1	DRSGroomD		X			Discharge
1	DRSMotA		X			Admission
1	DRSMotD		X			Discharge
1	DRSToiletA		X			Admission
1	DRSToiletD		X			Discharge
1	DRSVerA		X			Admission
1	DRSVerD		X			Discharge
FIM						
1	FIMCOGA	Calculated	X			Admission
1	FIMCOGD	Calculated	X			Discharge
1	FIMMOTA	Calculated	X			Admission

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	FIMMOTD	Calculated	X			Discharge
1	FIMTOTA	Calculated	X			Admission
1	FIMTOTD	Calculated	X			Discharge
1	FIMCompA		X			Admission
1	FIMCompD		X			Discharge
1	FIMExpressA		X			Admission
1	FIMExpressD		X			Discharge
1	FIMMemA		X			Admission
1	FIMMemD		X			Discharge
1	FIMProbSlvA		X			Admission
1	FIMProbSlvD		X			Discharge
1	FIMSocialA		X			Admission
1	FIMSocialD		X			Discharge
GCS						
1	GCSCat	4 Categories		X		
1	GCS	Calculated	X			
1	GCSTot		X			
1	GCSEye		X			
1	GCSMot		X			
1	GCSVer		X			
Severity						
1	TFCDays	Calculated	X			
1	PTADays	Calculated	X			
1	PTAdate					
1	PTAMethod					
Pre-Injury Conditions						
1	PreconBlind	Binary				
1	PreconDeaf	Binary				
1	PreconPhys	Binary				
1	PrelimDress	Binary				
1	PrelimLearn	Binary				
1	PrelimOuthm	Binary				

Form 1 or 2	Variable	Type	Continuous	Categorical Variables to be Collapsed	Variables to be Dummy Encoded	Variables that Change Over Time
1	PrelimWork	Binary				

Calculated Variables

Calculated variables are defined as those that are created via a formula using other variables as inputs. These are distinct from the TBIMS Data Dictionary definition, which includes any variable that has been created via any form of variable manipulation. Below is a list of all variables in the TBIMS Data Dictionary which meet the white paper definition of "calculated variable." This list does not include variables that are subtractions of dates.

Calculated Variable	Input Variables
B3TCOMP, B3TCOMPF	All BTACT item-level variables
B3TEF, B3TEFF	All BTACT executive functioning item-level variables
B3TEM, B3TEMF	All BTACT episodic memory item-level variables
BMI	Height, Weight
All DRS total scores	All DRS item-level variables
FIMCOGA, FIMCOGD, FIMMOTA, FIMMOTD, FIMTOTA, FIMTOTD, FIMCOGF, FIMMOTF, FIMTOTF	All FIM item-level variables
GAD7TOTF	All GAD7 item-level variables
GCS	All GCS item-level variables
GOSEF	All GOSE item-level variables
Malec_Prod	PRTWork, PRTSchool, PRTHome
Malec_Social	PRTSocFrnd, PRTSocFam, PRTEmotSup, PRTInternet
PART_BalancedF	PARTOutAbout, PARTProductivity, PARTSocial, PARTSummary
PART_Domain_OutF	PARTOutAbout, PARTSummary
PART_Domain_ProdF	PARTProductivity, PARTSummary
PART_Domain_SocF	PARTSocial, PARTSummary
PART_SDF	PARTOutAbout, PARTProductivity, PARTSocial, PARTSummary
PARTOutAboutF	PRTOutHse, PRTEatOut, PRTShop, PRTPlaySport, PRTMovie, PRTWtchSport, PRTReligion
PARTProductivityF	PRTHome, PRTSchool, PRTWork
PARTSocialF	PRTSocFrnd, PRTSocFam, PRTEmotSup, PRTInternet, PRTSpouse, PRTRelation, PRTFriend
PARTSummaryF	PARTOutAbout, PARTProductivity, PARTSocial
PHQ9TOTF	All PHQ9 item-level variables

Calculated Variable	Input Variables
PROBLEMUse, PROBLEMUseF	Drugs, ALC5Drinks, DrinkCat
SWLSTOTF, SWLSTOT4F	All SWLS item-level variables
TBIMS_NSDI_2019	PercentUnemployed, PercentSingleHoH, PercentNoHSorGED, PercentBSorUp, PercentBelowPoverty, PercentSNAP, MedHHIncome, MedFamIncome