ID5059 - Imputation

Python and R Hybrid

How can we use both? The `reticulate` library (2020) enables us to use both Python and R functions and coding in one place, a R Markdown file.

More information on the library specifications and functionalities can be found here

```
library(reticulate) # Allowing to use Python code in RMarkwdown.
```

All other requirements for this document:

```
# For general use
library(dplyr)          # Allows to use '%>%' and handy functions that help with data wrnagling.
library(tidyr)          # Allows to use method 'gather()'

# For Analysis
library(missForest)     # For multivariate methods imputers
library(mice)           # For multivariate methods imputers

# For Performance Metrics
library(caret)          # For confusion matrix
library(matrixStats)    # For calculating sd of multiple columns
```

Conducting Simulation Study of Imputation Methods

References

References for this document

1 : Flexible Imputation of Missing Data - Second Edition - Stef Van Buuren

2 : Inference and Missing Data - Rubin

3 : MICE: Multivariate Imputation by Chained Equations in R

4 : MissForest—non-parametric missing value imputation for mixed-type data

5 : Imputing Missing Data with R MICE Package

6 : MissForest : Non-Parametric imputation technique

Imputation

Imputation is the act of converting an incomplete sample into a complete sample. [1]

The idea behind this analysis is to construct a simulation study. This traditionally can be split into three key steps:

1. Simulation Step
2. Application of Methods
3. Assessing Performance of Methods

---

1. Removing Data At Random [Simulation Step]

Types of Missing Data

Rubin [2] classified missing data problems into three categories, which are all driven by the so-called **response mechanism**. This mechanism can be modelled, as long as an assumption is made as to which response mechanism drove the data to be missing in the first place. The categories are **Missing Completely at Random** (MCAR), **Missing at Random** (MAR), **Missing not at Random** (MNAR). In short, these can be described as;

1. **MCAR** : when the probability of data missing is the same across the entire datset.
2. **MAR** : when the probability of data missing is the same across groups within the dataset.
3. **MNAR** : when the probability of data missing depends on an unknown factor.

The current **methodologies** in the literature rely on a **key assumption** choice of either three categories. Hence, certain methods will be more suitable than others depending on the data and whether the assumption is met or not will underpin the validity of statistical inferences.

Another important distinction to make it between **Univariate** and **Multivariate Missing Data**. These are defined as the problem where only one feature is missing and the problem where more than one feature is missing respectively. In our simulation function we will have freedom to choose every time which case to be in.

In Practice

For this special case, the simulation step is slightly unusual. Normally one would sample from a known distribution, however in this instance the sampling happens as way to **identify the location of the values which will be dropped**, i.e. become NaN.

1. For the **MCAR** Case:

Mathematically, let n be the total number of observations (rows) in the dataset and m be the total number of features (columns) in the dataset. Hence, let U_1 and U_2 be two sets $U_1 \in 1, ..., n$ and $U_2 \in 1, ..., m$. By sampling with replacement k times from these two sets independently, such that any outcome is equally likely, the resulting ordered outcomes indicate the indexing of the element which will be dropped. These indexes can be viewed as the tuples (u_{1i}, u_{2i}) for $i \in 1, ..., k$.

Function to Remove Data (in Python)

```python
def remove_random_values(true_df, columns, percentage):
    """
    Funtion the will remove values from the selected columns at random,
    then replace the removed values with np.nan which is NaN.

    PSEUDOCODE:

    1. Random sample row indexes (with replacement)
    2. Random sample column labels (with replacement)
    3. Extract random entry using 1. and 2. and replace with np.nan
    4. Repeat (1-3) until total proportion of data has been extracted

    :param df: pandas DataFrame which is the full data
    :param columns: list of columns to 'drop' values from. Note: must be string name of columns
    :param percentage: float input such as 0.20 (i.e. 20%)
    :return: pandas DataFrame of the same shape as the original with NaN entries
    """

    # Set up: Focus on subset of data dependent on columns,
    #         hence extract information about subset and number of values to drop
    subset = true_df[columns].copy()
    nrow, ncol = true_df[columns].shape
    n_samples = int(percentage*nrow)

    # 1. Pick out a vector of random samples (row indexes)
    row_indexes = np.arange(0, nrow, 1).tolist()

    # 2. Sample with replacement from the list of possible indexes
    #    note, `choices` is used to sample with replacement
    sampled_row_indexes = random.choices(row_indexes, k = n_samples)
    sampled_columns = random.choices(columns, k = n_samples)

    # 3. Extract entry and replace accordingly
    for i in range(0, n_samples):
        if sampled_columns[i][0:3] == 'cat':
            subset.loc[sampled_row_indexes[i], sampled_columns[i]] = 'nan'
        else:
            subset.loc[sampled_row_indexes[i], sampled_columns[i]] = -1
            # this is the convention for a number of sklearn packages

    # Notes: having it as a loop ensures we are not holding a massive dataframe in memory so its a lot
    # Notes: must use 'nan' and numerical as np.nan would not be updated in step 4. otherwise.

    # 4. Update data frame with new NaN values
    true_df.update(subset)
    true_df = true_df.replace('nan', np.nan)
    true_df = true_df.replace(-1, np.nan)

    return(true_df)
```

2. Choosing the Imputation Methods [Methods Step]

Types of Models

The simplest and most common solutions for missing data problems are

- *likewise deletion*, which is simply removing the rows with missing elements from the dataset,
- *mean* or *median imputation*, which is substituting the missing feature with the sample mean or median over all observed values in that same feature column.
- *regression imputation*, which is about creating a regression model, hence incorporating the effects of other features, and using the model to predict the missing data.
- *stochastic regression imputation*, which is similar as above but incorporates variability of the predictions.
- *and more...*

In general however, these fall into two broad categories: **Univariate** and **Multivariate Imputation Methods**. This simply indicates whether the method takes into account all the data features to determine the missing data (multivariate) or only the column the missing data belongs to (univariate).

There is a key increase in **computational effort when using multivariate methods**, however these often perform better as they take into consideration the relationship between all the data when making a choice. This implies that often **correlation structures** across features are kept intact, compared to univariate methods which completely disregard them.

For the sake of this analysis and simulation study, we will explore both in some level of detail.

Due by **multivariate nature of the sampling** *(i.e. dropping data from more than one feature)*, there are three main strategies to consider:

1. **Monotone data imputation**. Imputations are created by a sequence of univariate methods;

2. **Joint modeling**. Imputations are drawn from a multivariate model fitted to the data;

3. **Fully conditional specification**, also known as multivariate imputation by chained equations. A multivariate model is implicitly specified by a *set of conditional univariate models*. Imputations are created by drawing from iterated conditional models.

   Note : 1. is better suited for monotone missing data patterns and 2. and 3. are suitable for general missing data patterns.

Univariate Methods to consider:

Univariate imputation methods chosen fall into some of the standard ideas which were mentioned before.

- `Median SimpleImputer`, this is a standard sklearn imputer which takes the column feature and uses the median value as the imputed value all missing values. *Note*: this only copes with numerical features.

- `MostFrequent SimpleImputer`, this is another standard sklearn imputer which takes the column feature and uses the most frequent category as the imputed value for all missing values. *Note*: this only copes with categorical features.

Function to Run Univariate Imputation Methods

```python
def univariate_imputation_method(nans_df):
    """
    Function which runs through our chosen three methods: Median Imputation, and Most Frequent Imputati

    Code from https://dzone.com/articles/imputing-missing-data-using-sklearn-simpleimputer helped remov
    when imputing isolated columns alone. Was getting error 'expected 2D array and was provided with 1D

    :param nans_df: pandas DataFrame which has missing data - subset on the columns to impute only
    :return: the imputed dataset
    """

    # Dataset that will be imputed and returned to main
    imputed_data = nans_df.copy()

    # Looping through each column that has to be imputed.
    # Doing this tactic to capture name of column
    for column in imputed_data:
        # Checking if the column is categorical data type and using most frequent tactic
        if column[0:3] == 'cat':
            imputer_tactic = SimpleImputer(strategy = "most_frequent")
        # Checking if the column is numerical data type and using mean tactic
        else:
            imputer_tactic = SimpleImputer(strategy = "mean")

        # Imputing the specific column with the appropriate specific tactic
        imputed_data[column] = imputer_tactic.fit_transform(imputed_data[column].values.reshape(-1,1))[

    return imputed_data
```

Multivariate Methods to consider:

For this analysis we focused on two key approaches.

- `MICE`, Multivariate Imputation by Chained Equations. More info can be found in the docs
- `MissForest` Imputer, this is a Nonparametric Missing Value Imputation using **Random Forest**. More info can be found in the docs

  Note : both cope with mixed data well.

MICE Fitting and Function

One of the key assumptions of MICE is that the **response mechanism is MAR**. Hence, that there is value in imputing values using a combination of all other features of the data. Which features are used as the *predictive variables* for the *imputed values* is shown in the output of PredictiveMatrix.

> For datasets containing hundreds or thousands of variables, using all predictors may not be feasible (because of multicollinearity and computational problems) to include all these variables. It is also not necessary. [1]

The **recommended steps for large datasets**, in short, are:

1. Include all features that appear in the model that will be applied to the data after imputation, including the outcome.

2. Include any feature which is known to be likely to be a driver for the missing data.
3. Include any feature which seems to explain a lot of the variance of the data.

Note : The mice package contains several tools that aid in automatic predictor selection. The `quickpred()` function is a quick way to define the predictor matrix using the strategy outlined above. [3] … The mice() function detects multicollinearity, and solves the problem by removing one or more predictors for the model. Each removal is noted in the loggedEvents element of the mids object.

Lastly, there is a wide range of choices in terms of the `methods` to deploy for this multivarite fitting. More information can be found at [1] - *Section 6.3 Model form and predictors.*

The **default methods used** are:

- for unordered factor variables with more than two levels : `polyreg` - Bayesian Polytomous Regression
- for continuous variables : `ppm` - Predictive Mean Matching

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

### FUNCTION
# Function that imputes missing values using the 'mice' machine learning algorithm.
## INPUT
# nans_df: data frame containing all NA values.
# columns: which columns to be considered in the imputation prediction.
## OUTPUT
# (data frame's) that data was imputed with mice. According to the variable m is the number of data fra
impute_using_mice <- function(nans_df, columns) {

  imp_df_mice <- mice(nans_df[columns], m = 3, maxit = 5, print = FALSE)

  return(imp_df_mice)

}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Notes;

- Interestingly, when running the MICE imputer on the categorical data only, this error comes up:

  `# Error in edit.setup(data, setup, ...) : mice detected constant and/or collinear variables. No predictors were left after their removal`

  Which indicates to me that all categorical variables may be drawn from the exact same sampling and that the labelling has no true meaning. Hence, for some meaningful predictions I kept a continuous feature `cont0` as the main predictor for both categorical features to impute.

MissForest Fitting and Function

The machine learning imputation method 'MissForest' follows an iterative lifecycle and is **based on the 'Random Forest' predictive model**. The specific algorithm does not fill missing cells with constants but with predictions. It predicts the missing values using a 'Random Forest' classification or regression model and the final predicted cell value is the **average or the majority of predictions obtained**.

- Average used when we filling numerical data types.
- Majority is used when we filling categorical data types.

The steps and process of a **MissForest lifecycle**:

1. Fill missing values using the mode ('most frequent') or mean of the column.
2. Train a Random Forest model using a complete data set.
3. Predict the missing values using the model trained above.
4. Re-fit Random Forest with dataset filled with the previous predicts values rather than mean/mode
5. Predict missing values again.
6. The above process is done in an iterative fashion and stops according to the stopping technique used

The MissForest imputation technique has outperformed other methods multiple times in different researches. Using the Random Forest models allows to take the advantage of out of **bag sample evaluating technique**, which does not need a test set. These techniques help and allow the algorithm to **perform very well when the data follows a complex non linear relation** within it. However when it follows a linear relation, other models are preferred which are created to capture such relations. Another benefit of using the MissForest it that it considers the randomness of data and is computationally efficient, which allows to deal with data of high dimensions. In addition to, the imputation method can deal with categorical and numerical data types. When predicting a categorical data type, classification tree models are used in the forest. On the other hand when predicting a numerical data type, regression tree models are used in the forest.

The main benefit of using a MissForest model is that the algorithm is non-parametric. It does not assume any assumptions on the structural aspects of the data. This has no constraint of when a MissForest is appropriate, as it can deal with structured data and unstructured data. More on the MissForest algorithm can be found on reference [6]

We can apply the MissForest imputation method using R package 'missForest' provided for us.

```r
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

### FUNCTION
# Function that imputes missing values using the 'missForest' machine learning algorithm.
## INPUT
# nans_df: data frame containing all NA values.
# columns: which columns to be considered in the imputation prediction.
## OUTPUT
# (data frame) that data was imputed with missForest
impute_using_missForest <- function(nans_df, columns) {

  imp_df_forest <- missForest(nans_df[columns], maxiter = 5)$ximp

  return(imp_df_forest)

}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Notes;

- By construction of algorithm cannot run on categorical variables with more than 53 factors: drop cat10 and cat8 and cat5. There are different techniques to deal with this phenomeno. We can apply a One v One Strategy but this would make the model more computationl expensive. Since we have many multiple variables, we choose to not include them in the Random Forest and use the rest.

3. Evaluating Performance of the Imputation Methods [Evaluation Step]:

Types of Adequate Performance Metrics:

For Continuous Features:

- **Normalized Root Mean Squared Error (NRMSE)**:
  $\sqrt{mean((X_{true} - X_{imp})^2)/var(X_{true})}$
  where X_{true} the complete data matrix and X_{imp} the imputed data matrix
  For good performance we expect values to be close to zero.
- **Difference between true value and imputed value**
  $D = x_{true} - x_{imp}$
  where x_{true} the true value from the data matrix and x_{imp} the imputed value from the data matrix
  For good performance we expect values to be concentrated about zero.
- **Percentage difference between true value and imputed value**
  $PD = 100 * |(x_{true} - x_{imp})/x_{true}|$
  where x_{true} the true value from the data matrix and x_{imp} the imputed value from the data matrix
  For good performance we expect values to be concentrated close to zero.

For Discrete Features:

- **Confusion Matrix** : count the number of *True Positives*, *False Negatives*, *False Positives* and *True Negatives* and store them in a matrix to assess performance of predictor.
  Good performance is indicated by higher counts in the diagonal of the matrix.
- **Accuracy** : sum the counts in the *diagonal* of the confusion matrix (True Positive and True Negative) and divide by the sum of all entries in the confusion matrix.
  Good performance leads to a value close to 1 and bad performance to a value around 0.

Functions to Evaluate Performance Metrics (Python)

```python
def performance_results_py(true_df, imp_df, nans_df, columns):
    """
    Function to extract true values and imputed values and compute performance metrics.
    :param true_df: pandas DataFrame containing true values
    :param imp_df: pandas DataFrame containing imputed values
    :param nans_df: pandas DataFrame containing nans
    :param columns: columns containing imputed values
    :return: list of results such that indexing returns results for each column (in columns) respective
             for each column you then have the output from performance_metrics() which depends on the d
    """

    results = []
    for column in columns:

        # a. Extract boolean vector (True/False) to identify Imputed True Data Vectors
        # Expect different values in cell, because comparing true with the NaN removed.
        boolean_values = (true_df != nans_df)[column].to_numpy().tolist()

        # c. Extract Imputed Data Vectors
        # Using Boolean Pointer list from above, extract true values
        true_values = true_df[column][boolean_values]
        # Using Boolean Pointer list from above, extract imputed values
        imputed_values = imp_df[column][boolean_values].to_numpy().tolist()

        # d. Evaluate performance metrics
        # Call function the two columns and obtain metrics.
        results.append(performance_metrics(imputed_values, true_values))

    return(results)

def performance_metrics(imputed_values, true_values):
    """
    Function to calculate all performance metrics of interest for each list of imputed vs true values.

    :param imputed_values: list of imputed values
    :param true_values: list of true values
    :return: if categorical - measures of accuracy and the confusion matrix
             if numerical - measures of raw difference and percentage difference between
                            imputed value and true value

    Note: each list for now is specific to each column, as it supports univariate imputations.
    Note: depending the on the methods used the columns are independent of each other,
          e.g. the median imputation does not use any other column at all -
          implying that we can assess performance independently of one another.
    """

    # If Categorical Feature
    if isinstance(imputed_values[0], str):

        # Confusion Matrix
        conf_matrix = metrics.confusion_matrix(true_values, imputed_values)

        # Accuracy
        accuracy = metrics.accuracy_score(true_values, imputed_values)

        return(conf_matrix, accuracy)

    # If Numerical Feature
    elif isinstance(imputed_values[0], float):

        # 1. The difference between the imputed value and truth
        D = np.array(imputed_values) - np.array(true_values)
        sample_mean = np.mean(D)
        sample_sd = np.sqrt(np.var(D))

        # 2. The percentage difference between the imputed value and truth
        PD = 100 * abs((np.array(imputed_values) - np.array(true_values))/np.array(true_values))
        # 3. Normalised Root Mean Squared Error
        NRMSE = sqrt(mean_squared_error(true_values, imputed_values)/np.var(true_values))

        return(D, PD, sample_mean, sample_sd, NRMSE)

    else:

        return(None)
```

Function to Evaluate Performance Metrics (R)

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## FUNCTION
#  Function to extract true values and imputed values and compute performance metrics for mice.
## INPUTS
#  true_df: data.frame containing true values
#  imp_df: output directly from methods `mice()`
#  columns: vector input containing strings representing column names which contain imputed values
## OUTPUT
#  list containing:
#    for categorical - measures of accuracy and the confusion matrix
#    for numerical - measures of raw difference and percentage difference between
#                    imputed value and true value
## NOTE
# number_iterations : Number of iterations mice has done.
# Each iteration computes a different imputation, we will find results from all imputations and get the
performance_results_mice <- function(true_df, imp_df, columns) {

  # Set up

  number_iterations <- ncol(imp_df$imp[[columns[1]]])
  all_results <- list()

  # For each column we imputed values we will found a metric.
  for (column in columns) {

    # 1. Extract True and Imputed

    # a) extract row indexes containing imputed values
    imp_rows <- row.names(imp_df$imp[[column]])

    # b) use row indexes to find true values
    truth <- data.frame(true_df[[column]])[imp_rows, ]

    # 2. Check if it is continuous or categorical values to apply appropriate computations.

    # a) If categorical column
    if (startsWith(column, "cat")) {

      accuracies <- c()

      for (i in 1:number_iterations) {
```

```r
        # extract imputed values for each iteration of mice.
        pred <- imp_df$imp[[column]][[i]]
        # compute metrics
        analysis_table <- table(pred, truth)
        res <- confusionMatrix(analysis_table)
        accuracies <- c(accuracies, res$overall[1])
      }

    all_results$conf_matrix[[column]] <- res$table
    all_results$accuracy[[column]] <- mean(accuracies)

    # b) If numerical column
    } else {

    all_results$D[[column]] <- truth - imp_df$imp[[column]][,]
    all_results$PD[[column]] <- 100 * abs((truth - imp_df$imp[[column]][,]) / truth)
    all_results$mean[[column]] <- colMeans(all_results$D[[column]])
    all_results$sd[[column]] <- colVars(as.matrix(all_results$D[[column]]))

    nrmse <- c()
    for (i in 1:number_iterations) {
      # extract imputed values for each iteration of mice.
      pred <- imp_df$imp[[column]][[i]]
      # compute metrics
      nrmse <- c(nrmse, sqrt((RMSE(pred, truth))^2/var(truth)))
    }

    all_results$nrmse[[column]] <- mean(nrmse)

    }
  }

  return(all_results)

}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## FUNCTION
#  Function to extract true values and imputed values and compute performance metrics for missForest.
## INPUTS
#  true_df: data.frame containing true values
#  imp_df: output directly from methods `missForest().ximp`
#  nans_df: data.frame which we randomly replaced true values with NA's
#  columns: vector input containing strings representing column names which contain imputed values
## OUTPUT
#  list containing:
#    for categorical - measures of accuracy and the confusion matrix
#    for numerical - measures of raw difference and percentage difference between
performance_results_missForest <- function(true_df, imp_df, nans_df, columns) {

  all_results <- list()
  # For each column we imputed values we will find a metric.
  for (column in columns) {

    # 1. Extract True and Imputed

    # a) extract true values to compare with imputed.
    rows_imputed <- nans_df %>%
      mutate(row_names = row.names(nans_df)) # Creating a row_id in data frame.

    # b) Keeps rows that have NA values in column only.
    rows_imputed <- rows_imputed[is.na(rows_imputed[[column]]), ]

    # c) Extract the cells for true and imputed values to compare between.
    truth <- data.frame(true_df[[column]])[rows_imputed[["row_names"]], ]
    pred <- data.frame(imp_df[[column]])[rows_imputed[["row_names"]], ]

    # 2. Check if it is continuous or categorical values to apply appropriate metrics.

    # a) If categorical column
    if (startsWith(column, "cat")) {

      accuracies <- c()

      analysis_table <- table(pred, truth)
      res <- confusionMatrix(analysis_table)
      accuracies <- c(accuracies, res$overall[1])

      all_results$conf_matrix[[column]] <- res$table
      all_results$accuracy[[column]] <- mean(accuracies)

    # b) If numerical column
    } else {

    all_results$D[[column]] <- truth - pred
    all_results$PD[[column]] <- 100 * abs((truth - pred) / truth)
    all_results$mean[[column]] <- mean(all_results$D[[column]])
    all_results$sd[[column]] <- sd(as.matrix(all_results$D[[column]]))
    all_results$nrmse[[column]] <- sqrt((RMSE(pred, truth))^2/var(truth))

    }
  }

  return(all_results)

}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Function to Store Performance Metrics (R)

```r
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## FUNCTION
```

```r
#  Function to extract all results for statistical testing in one dataframe
## INPUTS
#  simple_imputer_result: results from fitting
#  mice_result: results from fitting
#  missforest_result: results from fitting
#  columns: vector input containing strings representing column names which contain imputed values
## OUTPUT
#  dataframe
store_performance_results <- function(simple_imputer_result, mice_result, missforest_result, columns) {

  final_results <- c()

  i = 0
  # Loop through each column to store respective results of interest
  for (column in columns) {
    i = 1 + i

    if (startsWith(column, "cat")) {
      # - - - - For Categorical Columns

      accuracy <- c(simple_imputer_result[[i]]$accuracy, mice_result$accuracy[[i]], missforest_result$ac

      accuracy_results <- data.frame("performance" = accuracy,
                                     "performance_type" = rep("accuracy", each = 3),
                                     "columns" = rep(column, each = 3),
                                     "model" = c("simple",  "mice", "missforest"),
                                     "type" = rep("cat", each = 3))

      final_results <- rbind(final_results, accuracy_results)

    } else {
      # - - - - For Numerical Columns

      if (startsWith(columns[1], "cat") & startsWith(columns[2], "cont")) {
        i = 1 # little fix for when we have one cat and one cont columns to impute as indexing is diffe
        nrmse <- c(simple_imputer_result[[i + 1]]$nrmse, mice_result$nrmse[[i]], missforest_result$nrmse
      } else {nrmse <- c(simple_imputer_result[[i]]$nrmse, mice_result$nrmse[[i]], missforest_result$nrn

      nrmse_results <- data.frame("performance" = nrmse,
                                  "performance_type" = rep("nrmse", each = 3),
                                  "columns" = rep(column, each = 3),
                                  "model" = c("simple", "mice", "forest"),
                                  "type" = rep("cont", each = 3))

      final_results <- rbind(final_results, nrmse_results)

    }

  }

  return(final_results)
}
```

```r
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## FUNCTION
#  Function to extract all mean difference results for statistical testing in one dataframe
## INPUTS
#  simple_imputer_result: results from fitting
#  mice_result: results from fitting
#  missforest_result: results from fitting
#  columns: vector input containing strings representing column names which contain imputed values
## OUTPUT
#  dataframe
store_mean_diff_results <- function(simple_imputer_result, mice_result, missforest_result, columns) {

  # 1. IF scenario {both}
  if (startsWith(columns[1], "cat") & startsWith(columns[2], "cont")) {

    mean_diff <- c(simple_imputer_result[[2]]$mean, mice_result$mean[[1]], missforest_result$mean[[1]])

    mean_diff_results <- data.frame("mean_diff" = mean_diff,
                                    "model" = c("simple", rep("mice", 3), "forest"))

  # 2. IF scenario {cont_only}
  } else if (startsWith(columns[1], "cont") & startsWith(columns[2], "cont")) {

    mean_diff_col_1 <- c(simple_imputer_result[[1]]$mean, mice_result$mean[[1]], missforest_result$mean
    mean_diff_col_2 <- c(simple_imputer_result[[2]]$mean, mice_result$mean[[2]], missforest_result$mean

    mean_diff_results <- data.frame("mean_diff" = c(mean_diff_col_1, mean_diff_col_2),
                                    "model" = rep(c("simple", rep("mice", 3), "forest"), 2))

  }

  return(mean_diff_results)

}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Note: need to include data wrangling functions to transfer data from python to R

```r
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## FUNCTION
#  Function wrangles the data in appropriate structure for machine learning imputations.
## INPUTS
#  nans_df: data.frame containing random NA's values
## OUTPUT
#  data frames that are in appropriate state.
wrangle_data_for_na <- function(nans_df) {

  # Conversion issue between NaN and NA, hence we can explicitly use
  nans_df[nans_df == 'NaN'] = NA

  # Converting the data into R data frames
  nans_df <- as.data.frame(nans_df)

  # Unlist all of the columns
  columns <- colnames(nans_df)
  nans_df <- nans_df %>% mutate(across(all_of(columns), unlist))

  # All character types must be changed to factors
  nans_df <- nans_df %>% mutate_if(is.character, as.factor)

  return(nans_df)
}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

---

4. Bringing it all together

Need to make a loop over all possible scenario combinations.

First, load in data.

```
original_df <- py$read_true_values('data/train.csv')
```

Second, set up scenarios.

In order to set up the study, we made a few simplyfing assumptions.

- **Choose two columns to be our reference "predictor columns"** for all combinations of the simulation. This is due to how computationally expensive the fitting of the study is and by extension requires simplifying assumptions. By choosing a set of consistent predictor columns across the study, we control for any variability that could be introduced if these predictor columns changed at every iteration.

  Note, we chose a categorical and a numerical feature for the predictions, this is due to a multicollinearity issue in the prediction of multivariate imputation methods if we had only chosen one type.

```
columns_predictions_if_cat <- c('cont0')
columns_predictions_if_cont <- c('cat0')
columns_predictions_if_both <- c('cat0', 'cont0')
```

- **Block out four categorical variables from the study:** namely cat5, cat7, cat8 and cat10. This is because the number of levels of these categorical variables is over 50 and one of the imputation methods cannot handle with this dimensionality.

  Note, here we are also removing the predictor columns from the possible random picks of columns to be imputed.

```r
cat_vec <- 1:18
cat_vec <- cat_vec[!cat_vec %in% c(5, 7, 8, 10)]

cont_vec <- 1:10
```

- At this stage we did not focus on identifying predictive power of features and fine-tuning the imputation modelling for a specific set of features. Due to the randomness element of the study, we simply **choose the columns to be imputed at random**. The only requirement of the study is that you have the same number of occurrences of columns to be imputed which are {only continuous, only categorical, both}.

  This is controlled by looping through each combination and picking out a number of simulation as defined by `num_simulations`.

```r
combinations <- c("cont_only", "cat_only", "both")
num_simulations <- 3

i = 0
columns_combinations <- list()
for (combination in combinations) {
  for (iter in 1:num_simulations) {
    i = 1 + i

    if (combination == "cont_only") {

      cont_num <- sample(cont_vec, 2)
      columns_combinations[[i]] <- c(paste0('cont', cont_num[1]), paste0('cont', cont_num[2]))

    } else if (combination == "cat_only") {

      cat_num <- sample(cat_vec, 2)
      columns_combinations[[i]] <- c(paste0('cat', cat_num[1]), paste0('cat', cat_num[2]))

    } else if (combination == "both") {

      cat_num <- sample(cat_vec, 1)
      cont_num <- sample(cont_vec, 1)
      columns_combinations[[i]] <- c(paste0('cat', cat_num), paste0('cont', cont_num))

    }
  }
}
```

- **Choose a relevant set of percentages of data that should be dropped**: this is part of the set up of the simulation study and here we are exploring three cases $\{0.1, 0.2, 0.3\}$.

```r
percentage_drop <- list("large" = 0.3, "medium" = 0.2, "small" = 0.1)
```

- **Choose a subset of the data**: this is due to the computational expense of the runs complessively. In order to be able to run more scenarios in a local machine, we chose to take a subset of the data. This tradeoff was necessary due to the lack of access to a desktop computer and being unable to run this code overnight.

```
original_df <- original_df[0:100000, ]
```

Then, run the full simulation.

Third, full run.

```
final_results <- mean_diff_results <- c()

# Simulation Loops
i = 0
system.time(for (columns in columns_combinations) {
  for (percentage in percentage_drop) {

    i = 1 + i

    cat("Starting run", i)
    start_time <- Sys.time()
    cat(".")

    ### 1. Drop Values Randomly
    nans_df <- py$remove_random_values(original_df, columns, percentage)
    cat(".")

    ### 2. Simple Imputer
    simple_imputer <- py$univariate_imputation_method(nans_df[columns])
    simple_imputer_result <- py$performance_results_py(original_df, simple_imputer, nans_df, columns)
    cat(".", "\n")

    mid_time_1 <- Sys.time()
    cat("Simple Imputer Completed - Time Elapsed :", (mid_time_1 - start_time), '\n')

    ### 3. Data Wrangling (some Python to R conversions)

    # a) Appropriately change features of nans_df to R environment
    nans_df <- wrangle_data_for_na(nans_df)

    # b) Deal with Single Impute Results
    if (startsWith(columns[1], "cat") & startsWith(columns[2], "cat")) {
      names(simple_imputer_result[[1]]) <- names(simple_imputer_result[[2]]) <- c("conf_matrix", "accura
    } else if (startsWith(columns[1], "cont") & startsWith(columns[2], "cont")){
      names(simple_imputer_result[[1]]) <- names(simple_imputer_result[[2]]) <- c("D", "PD", "mean", "s
    } else {
      names(simple_imputer_result[[1]]) <- c("conf_matrix", "accuracy")
      names(simple_imputer_result[[2]]) <- c("D", "PD", "mean", "sd", "nrmse")
    }
    names(simple_imputer_result) <- c(columns[1], columns[2])

    ### 4. Multivariate Imputers

    # a) Set up prediction columns

    # IF {cat_only} scenario:
```

```r
    if (startsWith(columns[1], "cat") & startsWith(columns[2], "cat")) {
      pred_columns <- unique(c(columns, columns_predictions_if_cat))

    # IF {cont_only} scenario:
    } else if (startsWith(columns[1], "cont") & startsWith(columns[2], "cont")) {
      pred_columns <- unique(c(columns, columns_predictions_if_cont))

    # IF {both} scenario:
    } else { pred_columns <- unique(c(columns, columns_predictions_if_both)) }

    # b) MICE Imputer
    mice_imputer <- impute_using_mice(nans_df, pred_columns)
    mice_result <- performance_results_mice(original_df, mice_imputer, columns)

    mid_time_2 <- Sys.time()
    cat("MICE Imputer Completed - Time Elapsed :", (mid_time_2 - mid_time_1), '\n')

    # c) MissForest Imputer
    missforest_imputer <- invisible(impute_using_missForest(nans_df, pred_columns))
    missforest_result <- performance_results_missForest(original_df, missforest_imputer, nans_df, colum

    mid_time_3 <- Sys.time()
    cat("MissForest Imputer Completed - Time Elapsed :", (mid_time_3 - mid_time_2), '\n')

    ### 5. Store Results

    iter_results <- store_performance_results(simple_imputer_result, mice_result, missforest_result, co
    iter_results <- cbind(iter_results, "% drop" = c(rep(percentage, each = nrow(iter_results))))
    final_results <- rbind(final_results, iter_results)

    # IF {cont_only, both} scenarios:
    if (startsWith(columns[1], "cont") & startsWith(columns[2], "cont") | startsWith(columns[1], "cat")
      iter_results <- store_mean_diff_results(simple_imputer_result, mice_result, missforest_result, co
      iter_results <- cbind(iter_results, "% drop" = c(rep(percentage, each = nrow(iter_results))))
      mean_diff_results <- rbind(mean_diff_results, iter_results)
    }

    end_time <- Sys.time()
    cat("Time taken to compute full run", i, "for column combination", columns, "is :", (end_time - sta

  }
})
```

Save results!

```r
saveRDS(final_results, file = "imputation_results.rds")
saveRDS(mean_diff_results, file = "imputation_results_2.rds")
```

Analysing Results of Imputation Methods Performance

Necessary Libraries

```r
library(ggplot2)
library(dplyr)
library(RColorBrewer)
```

Read in results

```r
model_performance <- readRDS("imputation_results.rds")
mean_diff_estimates <- readRDS("imputation_results_2.rds")
```

Model Performance

Take a look at the model performances output

A snapshot of the results:

Given a total output of 162 observations due to all the possible combinations, a snapshot of this dataframe can be seen below:

```r
head(model_performance, 30)
```

```
##    performance performance_type columns  model type % drop
## 1    1.0000169            nrmse   cont9 simple cont    0.3
## 2    1.3945263            nrmse   cont9   mice cont    0.3
## 3    0.9841713            nrmse   cont9 forest cont    0.3
## 4    1.0000046            nrmse   cont5 simple cont    0.3
## 5    1.4108656            nrmse   cont5   mice cont    0.3
## 6    0.9990116            nrmse   cont5 forest cont    0.3
## 7    1.0002179            nrmse   cont9 simple cont    0.2
## 8    1.3902305            nrmse   cont9   mice cont    0.2
## 9    0.9822623            nrmse   cont9 forest cont    0.2
## 10   1.0000021            nrmse   cont5 simple cont    0.2
## 11   1.4002173            nrmse   cont5   mice cont    0.2
## 12   0.9908740            nrmse   cont5 forest cont    0.2
## 13   1.0000000            nrmse   cont9 simple cont    0.1
## 14   1.4196324            nrmse   cont9   mice cont    0.1
## 15   0.9723497            nrmse   cont9 forest cont    0.1
## 16   1.0000075            nrmse   cont5 simple cont    0.1
## 17   1.4072970            nrmse   cont5   mice cont    0.1
## 18   0.9895964            nrmse   cont5 forest cont    0.1
## 19   1.0000001            nrmse   cont3 simple cont    0.3
## 20   1.1137019            nrmse   cont3   mice cont    0.3
## 21   0.8041166            nrmse   cont3 forest cont    0.3
## 22   1.0000553            nrmse   cont7 simple cont    0.3
## 23   1.0238181            nrmse   cont7   mice cont    0.3
## 24   0.7579567            nrmse   cont7 forest cont    0.3
## 25   1.0000431            nrmse   cont3 simple cont    0.2
## 26   1.0848710            nrmse   cont3   mice cont    0.2
## 27   0.7946300            nrmse   cont3 forest cont    0.2
## 28   1.0000276            nrmse   cont7 simple cont    0.2
## 29   1.0107527            nrmse   cont7   mice cont    0.2
## 30   0.7322555            nrmse   cont7 forest cont    0.2
```

The measures of model performance chosen have been the:

- **Normalised Root Mean Squared Error** for the *Continuous Features*.

  Recall, the formula for the NRMSE is $\sqrt{mean((X_{true} - X_{imp})^2)/var(X_{true})}$, where X_{true} the complete data matrix and X_{imp} the imputed data matrix.

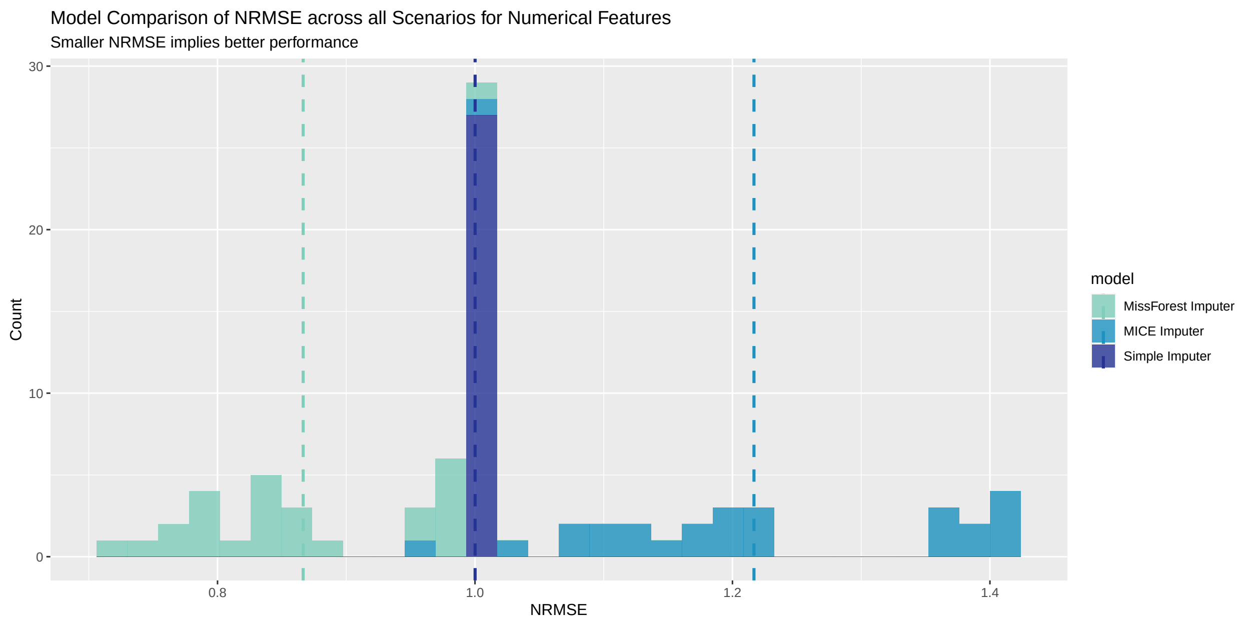
  In simple words, this metric is made up of the squared error over the deviation of the true underlying data, which normalises the error term and allows for meaningful comparisons across the different column estimates and scenarios. Moreover, for good performance we expect values to be close to zero.

```r
subset_model_performance <- model_performance[model_performance$type == 'cont', ]

# take group means, where the group of reference is the model
df <- data.frame(subset_model_performance %>% group_by(model) %>% summarize(mean = mean(abs(performance)

# set up labels names
labels <- c("MissForest Imputer", "MICE Imputer", "Simple Imputer")

# plot
ggplot(subset_model_performance, aes(x = abs(performance), fill = model)) +
  geom_histogram(bins = 30, alpha = 0.8) +
  geom_vline(data = df, aes(xintercept = mean, colour = model), linetype = "dashed", size = 1) +
  scale_color_manual(values = c(brewer.pal(9, "YlGnBu"))[c(4, 6, 8)], labels = labels) +
  scale_fill_manual(values = c(brewer.pal(9, "YlGnBu"))[c(4, 6, 8)], labels = labels) +
  labs(title = "Model Comparison of NRMSE across all Scenarios for Numerical Features",
       subtitle = "Smaller NRMSE implies better performance", x = "NRMSE", y = "Count")
```



```r
ggsave("nrmse_comparison.png", plot = last_plot())
```

Snapshot of the first 20 *results, sorted from best to worst,* which confirm what observed in the plot:

```
##     performance performance_type  model type % drop
## 36    0.7255462            nrmse forest cont    0.1
## 30    0.7322555            nrmse forest cont    0.2
## 24    0.7579567            nrmse forest cont    0.3
## 114   0.7680411            nrmse forest cont    0.3
## 120   0.7793170            nrmse forest cont    0.2
## 126   0.7806584            nrmse forest cont    0.1
## 33    0.7896211            nrmse forest cont    0.1
## 27    0.7946300            nrmse forest cont    0.2
## 21    0.8041166            nrmse forest cont    0.3
## 156   0.8292154            nrmse forest cont    0.2
## 54    0.8344071            nrmse forest cont    0.1
## 150   0.8347604            nrmse forest cont    0.3
## 162   0.8361582            nrmse forest cont    0.1
## 48    0.8455527            nrmse forest cont    0.2
## 42    0.8540660            nrmse forest cont    0.3
## 51    0.8614898            nrmse forest cont    0.1
## 45    0.8711523            nrmse forest cont    0.2
## 39    0.8785792            nrmse forest cont    0.3
## 144   0.9613986            nrmse forest cont    0.1
## 138   0.9663526            nrmse forest cont    0.2
```

The results clearly show that on average, as highlighted by the vertical lines, the **missforest imputation modelling performs best, followed by the simple imputer and the mice imputer** in this order. It appears that MissForest, even though it was computationally the most expensive, is a good choice for imputing continuous features.

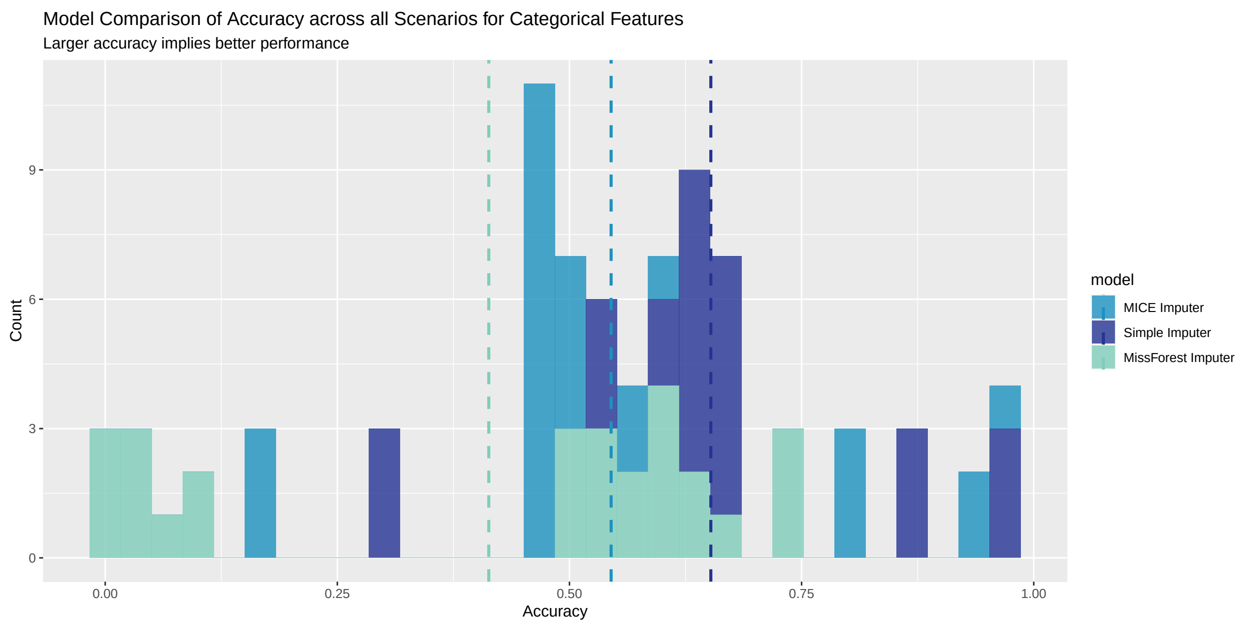
Moreover, the results in general seem *uneffected by the % of values dropped.*

Due to the multivariate nature of the imputation methods (missForest and MICE) it is apparent that the key driver behind the performance is how explanatory the predictor columns are for the randomly chosen columns to impute. Due to the key difference in assumptions between missForest and MICE, namely *non-parametric vs parametric* approach, it seems that the continuous features are better explained by non-linear relationships with the explanatory columns. Even though finding the explanatory columns was not of interest in this simulation study, it is still an interesting realisation of the results. The predictive power of the features will be explored in the later part of the project.

- **Accuracy** for the *Categorical Features*

  Recall accuracy is the sum the counts in the *diagonal* of the confusion matrix (True Positive and True Negative) and divide by the sum of all entries in the confusion matrix.

  In simple words, it captures how many imputed values were correctly identified over the total number of values which were imputed. Moreover, good performance leads to a value close to 1 and bad performance to a value around 0.



Snapshot of the first 20 *results, sorted from best to worst,* which confirm what observed in the plot:

```
##      performance performance_type      model type % drop
## 79     0.9758788         accuracy     simple  cat    0.2
## 73     0.9758059         accuracy     simple  cat    0.3
## 85     0.9755295         accuracy     simple  cat    0.1
## 74     0.9530363         accuracy       mice  cat    0.3
## 80     0.9519981         accuracy       mice  cat    0.2
## 86     0.9515388         accuracy       mice  cat    0.1
## 100    0.8581267         accuracy     simple  cat    0.2
## 94     0.8563692         accuracy     simple  cat    0.3
## 106    0.8552795         accuracy     simple  cat    0.1
## 101    0.7914812         accuracy       mice  cat    0.2
## 95     0.7913056         accuracy       mice  cat    0.3
## 107    0.7902692         accuracy       mice  cat    0.1
## 96     0.7375072         accuracy missforest  cat    0.3
## 108    0.7333333         accuracy missforest  cat    0.1
## 102   0.7318288         accuracy missforest  cat    0.2
## 127   0.6811948         accuracy     simple  cat    0.3
## 151   0.6807254         accuracy     simple  cat    0.2
## 145   0.6783013         accuracy     simple  cat    0.3
## 157   0.6751331         accuracy     simple  cat    0.1
## 133   0.6641438         accuracy     simple  cat    0.2
```

For accuracy results we observe a different outcome, with the **univariate imputer performing best, mice following in second place and missforest placing last**. This is likely due to the fundamental assumptions of the multivariate methods not being met by the data. The chosen predictor continuous covariate cont0 is likely not indicative of the behaviour of the randomly chosen categorical columns to impute.

This is a clear case of the strength of the simple imputer on average, given that we have a very complex structure of the data and finding the appropriate predictor columns can often be a challenge. In these instances where the relationship of the data is not immediately clear, it could be often recommended to rely on a univariate imputation method. In both scenarios, continuous and discrete, the simple imputer performs well overall.

Mean Difference Estimates

Take a look at the estimated output

A snapshot of this dataframe can be seen below

```r
head(mean_diff_estimates, 20)
```

```
##        mean_diff  model % drop
## 1  -0.0011299024 simple    0.3
## 2   0.0022754617   mice    0.3
## 3   0.0151230981   mice    0.3
## 4  -0.0028426041   mice    0.3
## 5   0.0023161684 forest    0.3
## 6   0.0007328600 simple    0.3
## 7   0.0008139346   mice    0.3
## 8   0.0012448486   mice    0.3
## 9   0.0045700727   mice    0.3
## 10  0.0071474805 forest    0.3
## 11  0.0040413478 simple    0.2
## 12  0.0257665885   mice    0.2
## 13 -0.0126167648   mice    0.2
## 14  0.0045644662   mice    0.2
## 15 -0.0112796256 forest    0.2
## 16 -0.0004913337 simple    0.2
## 17  0.0002166127   mice    0.2
## 18  0.0018650352   mice    0.2
## 19  0.0014348761   mice    0.2
## 20 -0.0025741345 forest    0.2
```

What do I mean by mean difference?

A good way to assess model performance in this instance, is to take a look at the **ability of each model to correctly impute the missing values**. This measure is obtained by taking the **average of the differences** between all the imputed values $\bar{x}_{\text{imp},i}$ for column i and all the true values $\bar{x}_{\text{true},i}$ for column i.

Hence: $MD_i = mean(\bar{x}_{\text{imp},i} - \bar{x}_{\text{true},i})$

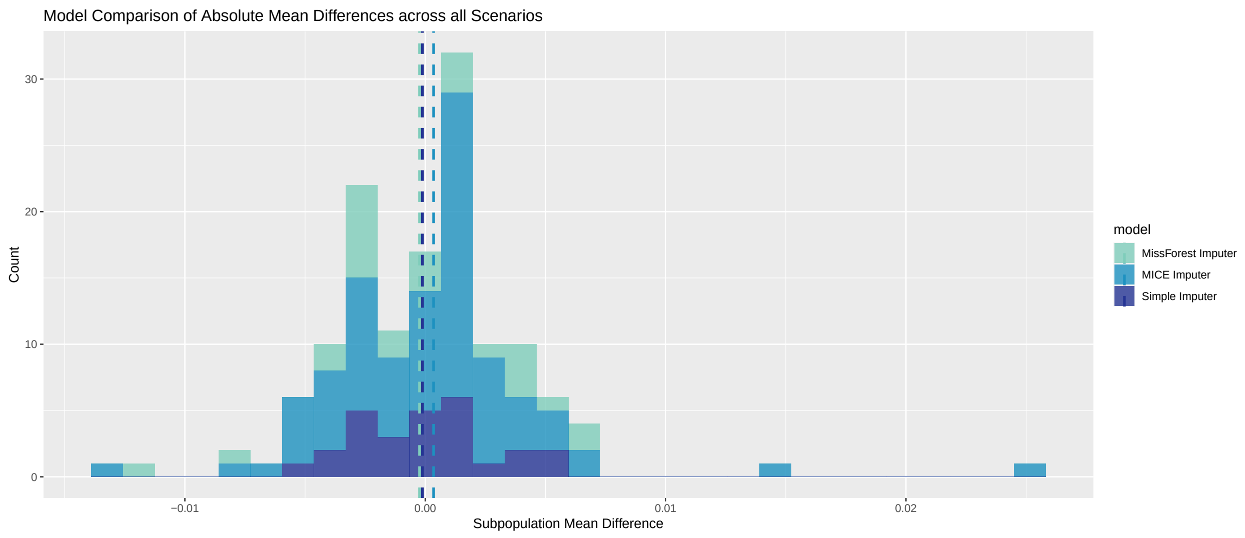
This measure is computed for each column of interest, where the column i was randomly chosen in the simulation as the column to drop values from, and for each scenario combination, where the scenario combinations are {cont_only, 0.1}, {cat_only, 0.1}, {both, 0.1}, {cont_only, 0.2}, … .

How do I make use of these measures?

First, one may wish to take a look at a group comparison. This can help highlight better performing models against poorer performing models in one go.

- For *poor performance* of a model, I expect to see the distribution of MD with an overall **mean further away from zero and a high variability**.
- For *good performance* of a model, I expect to see the distribution of MD with an overall **mean closer to zero and a smaller variability**.

Therefore;



This plot shows a similar information as the one represented by the NRMSE previously. Here, we have the distribution of the mean differences about zero, and it appears the models perform well with the missforest and the simple imputer head-to-head and the MICE performing worse than them. This confirms the conclusions drawn from the NRMSE metric

---

Limitations and Areas for Development

- **In terms of the limitations of the study and scenarios**:

  This simulation study captures *only a snapshot of the potential analysis over the imputation methods.* This is due to the simplifying assumptions that had to be made, because of the lack of more powerful equipment and the limited amount of time we had available to work on it.

It is important to note that the framework and the functions built in this study could support much more complex analysis and scenario building. However this could not be explored further for this report, for the same reasons mentioned above.

- **In terms of the models chosen**:

  We strongly believe that the models chosen cover a wide range of fundamental assumptions and are therefore suited for this analysis. This enabled us to draw some inference and evaluate likely causes for the poor performance of certain methods such as MICE. However, a more comprehensive way could have been to test the model assumptions first.

  Moreover, the 'prediction columns' were chosen arbitrarily and *another set of prediction columns could have led to a different outcome.* For this reason a more comprehensive analysis could have been to allow for the training of the multivariate methods on the entire dataset. If it wasn't for the computational limitations, this route could have been preferred.

- **In terms of the performance analysis**:

  We would have also liked to explore further the test statistic of the Difference, and carry out some statistical testing to improve the confidence of our results. One of the possible tests we considered was a **paired t-test** where the pair would be $(X_{\text{imp}}, X_{\text{true})}$, and the null hypothesis would be $H_0 : \mu_{X_{\text{imp}}} = \mu_{X_{\text{true}}})$.
  This test could have been repeated for each imputed column and could have highlighted the instances for which the imputed values diverged significantly from the true values, to the extent of having to reject the null hypothesis that the difference of the means is zero.