GAIN

Paolo Colussi

2024-11-08

```r
knitr::opts_chunk$set(echo = TRUE)

rm(list = ls())

library(reticulate)
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.0.4
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```r
library(readr)
library(keras)
library(tcltk)
library(tensorflow)
```

Introduction

This code provides an R implementation of **GAIN (Generative Adversarial Imputation Nets)**, a method for imputing missing data using a **Generative Adversarial Network (GAN)** framework. GAIN was introduced by Jinsung Yoon, James Jordon, and Mihaela van der Schaar in their paper **"GAIN: Missing Data Imputation using Generative Adversarial Nets."** The model consists of two key components: a generator, which predicts missing values in a dataset, and a discriminator, which attempts to distinguish between real and imputed values. The generator fills in the missing entries by identifying patterns in the observed data, while the discriminator evaluates the completed dataset to determine which values were originally missing.

The code is structured into several sections:

- *Utilities:* This section defines various functions required for the network's operation.
- *GAIN:* The core of the code, where the neural network is constructed and trained.
- *Implementation:* This section demonstrates two possible applications of GAIN in practice.
- *Test:* In this section the GAIN is tested.

Utilities

This section contains a set of utility functions that perform essential preprocessing and evaluation tasks for the GAIN algorithm.

missig

The `missig` function generates a mask to introduce missing values in a dataset. It takes three parameters as input: `p`, which represents the probability that a value is missing, `no`, which indicates the number of rows in the matrix to be generated, and `dim`, which specifies the number of columns. The output returns a mask matrix composed of no*dim elements made up of 1 and NA, specifically with a p proportion of NA to the total. We will use this function to artificially induce missingness in one of the two GAIN applications we have here.

```r
# This function generates a mask for introducing missing values into a dataset.
# Parameters:
# - p: Probability of a value being missing.
# - no: Number of rows in the generated matrix.
# - dim: Number of columns in the generated matrix.
missig <- function(p, no, dim) {
  # Create a matrix of random numbers between 0 and 1 with specified dimensions.
  unif_matrix <- matrix(runif(no * dim), no, dim)
  # Generate a mask where values are 1 with probability 'p', otherwise NA.
  mask <- ifelse((unif_matrix < p), 1, NA)
  return(mask)
}
```

find_mask

The `find_mask` function generates a mask based on the missingness in a dataset. It takes the dataset as input and returns a mask matrix composed by the same number of elements of the data, made of 1 and NA, basically transforming all the non-NA elements in the data in 1. This mask, similarly to the one calculated by the missing function, is needed to calculate the RMSE.

```r
# This function creates a mask identifying missing values in a dataset.
# Parameters:
# - data: Input dataset (matrix or dataframe).
# Returns:
# - A mask matrix where NA represents missing values and 1 represents observed values.
find_mask <- function(data) {
  mask <- ifelse(is.na(data), NA, 1)
  return(mask)
}
```

parameters_norm

The `parameters_norm` function calculates the minimum and maximum values of each column in a dataset, returning an array with these values. We will need this matrix for the normalization function, which performs a **min-max normalization.**

```r
# This function calculates the minimum and maximum values of each column in a dataset.
# Used to generate parameters for normalizing the data.
# Parameters:
# - data: Input matrix or dataframe.
parameters_norm <- function(data) {
  # Calculate the minimum of each column, ignoring NA values.
  min <- apply(data, 2, function(x) min(x, na.rm = TRUE))
  # Calculate the maximum of each column, ignoring NA values.
  max <- apply(data, 2, function(x) max(x, na.rm = TRUE))

  # Combine minimum and maximum values into a matrix with two columns.
  parameters <- cbind(min, max)
  return(parameters)
}
```

normalization

The `normalization` function applies **Min-Max normalization** to the data, scaling the values of each column to a range between 0 and 1.

It takes as input a matrix or dataframe (`data`) to be normalized and a matrix `parameters`, which contains the minimum and maximum values for each column (obtained from the `parameters_norm` function).

The normalization process occurs in two steps:

- The minimum value of the corresponding column is subtracted from each element.
- The result is then divided by the maximum value of the column, scaling the values to the range [0,1].

However, this is not the classic Min-Max normalization. In the standard approach, after subtracting the column's minimum, the values are divided by (max - min) instead of just max. The difference is that our normalization returns values in the range 0 to (max-min)/max, which becomes [0,1] only when the column's minimum is zero.

Why do we use this normalization? This question allows us to highlight the approach we took when writing this code: staying faithful to Yoon's original paper and the Python implementation of the network. This is, in fact, the same implementation found there.

```r
# This function normalizes the data using min-max normalization.
# Parameters:
# - data: Input matrix or dataframe to be normalized.
# - parameters: Matrix with min and max values for each column (from parameters_norm).
normalization <- function(data, parameters) {
  # Subtract the minimum value from each element (column-wise).
  data_norm <- sweep(data, 2, parameters[, 1], "-")
  # Divide by the range (maximum - minimum) for each column.
  data_norm <- sweep(data_norm, 2, parameters[, 2], "/")

  return(data_norm)
}
```

renormalization:

The `renormalization` function converts normalized data back to its original scale by reversing the Min-Max normalization process. It takes as input a normalized data matrix or dataframe (`data`) and a `parameters` matrix containing the minimum and maximum values for each column (obtained from the parameters_norm function). The renormalization process occurs in two steps:

- Each normalized value is multiplied by the column's maximum value.
- The column's minimum value is then added to each element, restoring the original scale.

However, this implementation assumes that the normalization was performed by dividing by max instead of (max - min), which is not the standard Min-Max normalization. If classic Min-Max normalization were used, the multiplication step should use (max - min) instead of max.

```r
# This function renormalizes the data, converting normalized values back to their original scale.
# Parameters:
# - data: Normalized data matrix or dataframe.
# - parameters: Matrix with min and max values for each column (from parameters_norm).
renormalization <- function(data, parameters) {
  # Multiply normalized values by the range (column-wise).
  data_renorm <- sweep(data, 2, parameters[, 2], "*")
  # Add the minimum value to each element (column-wise).
  data_renorm <- sweep(data_renorm, 2, parameters[, 1], "+")

  return(data_renorm)
}
```

rmse_loss

The `rmse_loss` function calculates the **Root Mean Squared Error (RMSE)** between the original dataset and the imputed dataset, considering only the missing values. It takes three inputs: `ori_data`, the original dataset assumed to have no missing values; `imputed_data,` the dataset after missing values have been imputed; and `data_m`, a mask matrix where 1 indicates observed values and NA represents missing values.

The function first normalizes both the original and imputed datasets using Min-Max normalization through the `parameters_norm` and `normalization` functions. Then, it processes the mask matrix by converting it into a dataframe and replacing NA values with 0 to properly identify missing entries.

To compute the RMSE, the function calculates the squared differences between the original and imputed values, focusing only on the missing entries. Mathematically, the RMSE is given by:

$$\text{RMSE} = \sqrt{\frac{\sum_{i,j}[(1 - M_{i,j})(X_{i,j} - \hat{X}_{i,j})]^2}{\sum_{i,j}(1 - M_{i,j})}}$$

where:
- $X_{i,j}$ is the original data value at position (i, j).
- $\hat{X}_{i,j}$ is the imputed data value at position (i, j).
- $M_{i,j}$ is the mask matrix, with $M_{i,j} = 1$ for observed values and $M_{i,j} = 0$ for missing values.
- The numerator sums the squared differences only for missing values.
- The denominator counts the total number of missing values.

Finally, the RMSE is obtained by taking the square root of this fraction.

```r
# This function calculates the RMSE between the original
# data and imputed data, focusing only on the missing values.
# Parameters:
# - ori_data: Original dataset (with no missing values).
# - imputed_data: Dataset after imputation of missing values.
# - data_m: Mask matrix indicating missing values (1 for observed, NA for missing).
rmse_loss <- function(ori_data, imputed_data, data_m) {
  # Normalize the original and imputed datasets.
  parameters <- parameters_norm(ori_data)
  ori_data <- normalization(ori_data, parameters)
  imputed_data <- normalization(imputed_data, parameters)

  # Replace NA values in the mask with 0.
  data_m <- as.data.frame(data_m) %>%
    mutate(across(everything(), ~replace_na(., 0)))

  # Calculate the squared differences, focusing only on missing entries.
  nominator <- sum(((1 - data_m) * ori_data - (1 - data_m) * imputed_data)**2)
  denominator <- sum(1 - data_m)  # Number of missing values.

  # Compute the RMSE value.
  rmse <- sqrt(nominator / denominator)

  return(rmse)
}
```

hint_matrix

The `hint_matrix` function generates a binary hint matrix that guides the imputation process by indicating which values are included as hints. The function takes three parameters: `hint_rate`, which represents the probability of including a value in the hint; `no`, the number of rows in the matrix; and `dim`, the number of columns. It outputs a no*dim matrix consisting of 1s and 0s, where the proportion of 1s in the total matrix is approximately equal to hint_rate. This function is used during the training of the network and is applied to each batch, from which a new hint matrix is generated. The hint matrix provides additional information to the discriminator about the mask, helping it distinguish between true and imputed values.

In Yoon's paper, this hint mechanism is shown to be crucial. If the hint does not contain sufficient information, the generator will be unable to learn the correct data distribution, making it impossible to impute the missing values effectively.

```r
# This function generates a "hint matrix," typically used in data imputation algorithms.
# The matrix is binary, indicating whether a value is included in the hint.
# Parameters:
# - hint_rate: Probability of including a value in the hint.
# - no: Number of rows in the generated matrix.
# - dim: Number of columns in the generated matrix.
hint_matrix <- function(hint_rate, no, dim) {
  # Create a probability matrix of random numbers between 0 and 1.
  prob_matrix <- matrix(runif(no * dim), no, dim)
  # Generate a binary hint matrix based on the hint rate.
  hint <- ifelse((prob_matrix < hint_rate), 1, 0)
  return(hint)
}
```

rounding

The `rounding` function adjusts the imputed dataset by rounding values in columns that likely represent categorical data. This is particularly useful when categorical variables have been normalized during preprocessing. The function takes two inputs: `imp_data`, the imputed dataset, and `or_data`, the original dataset, which is used to determine which columns require rounding. It iterates through each column of or_data and checks whether it has fewer than 20 unique values. If this condition is met, the corresponding column in imp_data is rounded to the nearest integer.

```r
# This function rounds imputed data for columns with fewer than 20 unique values.
# This is useful for categorical data that may have been normalized.
# Parameters:
# - imp_data: Imputed dataset.
# - or_data: Original dataset (to check the number of unique values in each column).
rounding <- function(imp_data, or_data) {
  for (i in 1:ncol(or_data)) {
    # Check if the column has fewer than 20 unique values.
    if (length(unique(or_data[, i])) < 20) {
      # Round the values in the imputed dataset for this column.
      imp_data[, i] <- round(imp_data[, i])
    }
  }
  return(imp_data)
}
```

GAIN

The `gain` function implements the **Generative Adversarial Imputation Network (GAIN)**, using a modified generative adversarial network (GAN). The function takes as input a dataset with missing values and generates an imputed version of the dataset by training a generator and a discriminator in an adversarial manner.

Theoretical Background

GAIN is a GAN-based approach to data imputation where a generator attempts to fill in missing values, while a discriminator tries to distinguish between observed and imputed values. The generator is trained not only to produce realistic imputations but also to fool the discriminator into classifying generated values as real. The discriminator receives additional information in the form of a hint matrix, which allows it to generalize beyond trivial patterns in the missingness.

The optimization is performed using two loss functions: - **Discriminator loss:** Measures how well the discriminator can classify real vs. imputed values. - **Generator loss:** Encourages the generator to both fool the discriminator and approximate the true data distribution.

Function Parameters and Their Roles

The function includes several parameters that control the training process:

- `data`: The input dataset containing missing values, which will be imputed by the model.
- `batch_size`: The number of samples used in each training batch, which affects the stability of optimization.

- `hint_rate`: The probability that a value is included in the hint matrix, which is used to provide partial information to the discriminator.
- `alpha`: A weighting factor that balances the loss term for fooling the discriminator and the Mean squared error (MSE) for observed data in the generator loss function.
- `iterations`: The number of training steps performed by the network.
- `learning`: The learning rate used for training both the generator and the discriminator, which affects convergence speed and stability.
- `loss_monitoring`: A flag that enables or disables loss tracking during training, allowing users to observe how the model improves over time.
- `beta_1` and `beta_2`: Parameters of the Adam optimizer, controlling momentum and stability during gradient updates.
- `initialization`: Specifies how the neural network weights are initialized, with options like **Xavier**, **Glorot**, or **Random**.
- `progress_bar`: Enables or disables a progress bar to visualize the training process.

Implementation

The function starts by defining the loss functions for both the discriminator and the generator. The discriminator loss is is trained to minimize the classification loss (when classifying which components were observed and which have been imputed). The generator loss is a combination of two terms, using the parameter `alpha`: 1. **Adversarial loss**: Encourages the generator to maximize the discriminator's misclassification rate. 2. **MSE loss**: Ensures consistency for observed values by using **mean squared error (MSE)**.

The dataset is then normalized using **min-max normalization**, with the `normalization` function, as we saw earlier. A mask matrix (`data_m`) is created to indicate which values are observed and which are missing. The normalization ensures that the generator operates within a stable numerical range.

Next, the **neural network architectures** for the generator and discriminator are defined. Both models use multiple dense layers with ReLU activation functions, followed by a final sigmoid activation to produce normalized outputs.

The **training loop** iterates for the specified number of steps (defined by `iterations`): 1. A batch of data is sampled, and missing values are replaced with small random noise. 2. A hint matrix is generated to provide the discriminator with partial information about the missing values. 3. The generator produces synthetic imputations, which are combined with observed data to create a completed dataset. 4. The discriminator is trained to classify observed vs. imputed values. 5. The generator is trained to minimize its loss and improve the quality of its imputations.

Throughout the training, the loss values are recorded to monitor the performance of both networks.

After training, the generator produces final imputations for the missing values. The imputed dataset is then renormalized (with the `renormalization` function) to its original scale and rounded (with the `rounding` function) for categorical variables where necessary.

The discriminator receives a partial hint on which values are imputed or not. This means that the discriminator has to classify only part of the element of the batch. As demonstrated in Yoon's paper, without an effective hint mechanism, the generator struggles to learn the true data distribution, leading to poor imputations. Tuning the `hint rate` is important to ensure that the discriminator receives just enough information to guide the generator.

Final Output

The function returns a list containing:

- `data`: The final imputed dataset.

- `D_loss`: A vector containing discriminator loss values recorded during training.
- `G_loss`: A vector containing generator loss values recorded during training.

This implementation follows the structure outlined in the original GAIN paper and aligns with the standard Python implementation.

```
# Function to perform data imputation using the GAIN algorithm
gain <- function(data,
                 batch_size,  # Number of samples per training batch
                 hint_rate,   # Probability of hint matrix values being revealed
                 alpha,       # Weighting factor for generator loss
                 iterations,  # Number of training iterations
                 learning,    # Learning rate for optimization
                 loss_monitoring,  # Enable or disable loss tracking during training
                 beta_1,      # Beta1 parameter for Adam optimizer
                 beta_2,      # Beta2 parameter for Adam optimizer
                 initialization, # Method for weight initialization (e.g., Xavier, Glorot)
                 progress_bar) {  # Enable or disable a progress bar

  # Define the loss function for the discriminator (D)
  D_loss <- function(y_true, y_pred) {
    M <- y_true[[1]]  # True mask matrix
    # Compute the discriminator loss using binary cross-entropy
    loss <- -k_mean(M * k_log(y_pred + 1e-8) + (1 - M) * k_log(1 - y_pred + 1e-8))
    return(loss)
  }

  # Define the loss function for the generator (G)
  # This consists of two parts:
  # - G_loss_temp: Encourages fooling the discriminator for missing values.
  # - MSE_loss: Enforces data consistency for observed values.
  G_loss <- function(y_true, y_pred) {
    M <- y_true[[1]]  # Mask matrix
    D_prob <- y_true[[2]]  # Output probabilities from the discriminator
    G_sample <- y_pred  # Generated data sample

    # Loss term for fooling the discriminator
    G_loss_temp <- -k_mean((1 - M) * k_log(D_prob + 1e-8))
    # Mean squared error (MSE) for observed data
    MSE_loss <- k_mean((M * X_mb - M * y_pred)^2) / k_mean(M)
    # Total generator loss
    G_loss <- G_loss_temp + alpha * MSE_loss
    return(G_loss)
  }

  # Generate the mask matrix indicating observed (1) and missing (0) values
  data_m <- 1 - is.na(data)

  # Store the dimensions of the data
  no <- nrow(data)
  dim <- ncol(data)

  # Normalize the data using min-max normalization
  norm_parameters <- parameters_norm(data)
```

```r
norm_data <- normalization(data, norm_parameters) %>%
  mutate(across(everything(), ~replace_na(., 0)))  # Replace NA with 0

# Define the input layers for the GAN
X <- layer_input(shape = c(dim))  # Data vector (input)
M <- layer_input(shape = c(dim))  # Mask vector (observed/missing indicator)
H <- layer_input(shape = c(dim))  # Hint vector (used to train discriminator)

# Define the discriminator architecture
inputs_D <- layer_concatenate(list(X, H))  # Combine data and hint vector

# Initialize weights using the chosen method
initializer <- switch(initialization,
                    "Xavier" = initializer_he_normal(),
                    "Glorot" = initializer_glorot_normal(),
                    "Random" = initializer_random_normal(),
                    stop("Invalid initialization method"))

# Build the discriminator network
D_output <- inputs_D %>%
  layer_dense(units = dim, activation = "relu",
              kernel_initializer = initializer) %>%
  layer_dense(units = dim, activation = "relu",
              kernel_initializer = initializer) %>%
  layer_dense(units = dim, activation = "sigmoid",  # Sigmoid for binary output
              kernel_initializer = initializer)

# Define the generator architecture
inputs_G <- layer_concatenate(list(X, M))  # Combine data and mask vector

# Build the generator network
G_output <- inputs_G %>%
  layer_dense(units = dim, activation = "relu",
              kernel_initializer = initializer) %>%
  layer_dense(units = dim, activation = "relu",
              kernel_initializer = initializer) %>%
  layer_dense(units = dim, activation = "sigmoid",  # Sigmoid for normalized output
              kernel_initializer = initializer)

 # Create the discriminator and generator models
discriminator <- keras_model(inputs = list(X, H), outputs = D_output)
generator <- keras_model(inputs = list(X, M), outputs = G_output)

# Compile the models with their respective loss functions
discriminator %>% compile(optimizer = optimizer_adam(learning_rate = learning), loss = D_loss)
generator %>% compile(optimizer = optimizer_adam(learning_rate = learning), loss = G_loss)

# Initialize a progress bar for training
if (progress_bar) pb <- tkProgressBar(title = "Progress", min = 0, max = iterations, width = 300)


# See below
 D_loss_values <- c()
 G_loss_values <- c()



# Training loop for the GAN
for (i in 1:iterations) {
  # Sample a batch of data
  batch_idx <- sample(1:no, batch_size)
  X_mb <- as.matrix(norm_data[batch_idx, ])  # Batch of normalized data
  M_mb <- data_m[batch_idx, ]  # Corresponding mask for the batch

  # Generate random noise and hint matrix for the batch
  Z_mb <- runif(batch_size * dim, min = 0, max = 0.01) %>% matrix(nrow = batch_size, ncol = dim)
  H_mb_temp <- hint_matrix(hint_rate, batch_size, dim)
  H_mb <- M_mb * H_mb_temp  + 0.5*(1 - H_mb_temp)

  # Combine observed data with random noise for missing values
  X_mb <- M_mb * X_mb + (1 - M_mb) * Z_mb

  # Generate data using the generator
  G_sample <- generator %>% predict(list(X_mb, M_mb), verbose=0)

  # Combine real and generated data
  Hat_X <- X_mb * M_mb + G_sample * (1 - M_mb)

  # Generate discriminator probabilities
  D_prob <- discriminator %>% predict(list(Hat_X, H_mb), verbose=0)

  # Train the discriminator on real and generated data
  D_loss_value <- discriminator %>% train_on_batch(list(X_mb, H_mb), M_mb)

  # Train the generator to improve its performance
  G_loss_value <- generator %>% train_on_batch(list(X_mb, M_mb), list(M_mb, D_prob))

  # # Add losses to the list for monitoring
   D_loss_values <- c(D_loss_values, D_loss_value)
   G_loss_values <- c(G_loss_values, G_loss_value)


  # print the losses every 100 iteration
  if (i %% 100 == 0 & loss_monitoring) cat("\rIteration", i,"\tGloss", G_loss_value, "\tDloss", D_loss

  # Update the progress bar
  if (progress_bar) setTkProgressBar(pb, i, label = sprintf("Progress: %d%%", round(i / iterations * 
}

# Close the progress bar after training
if (progress_bar) close(pb)

# Generate final imputed data
Z_mb <- matrix(runif(no * dim, min = 0, max = 0.01), nrow = no, ncol = dim)
X_mb <- data_m * norm_data + (1 - data_m) * Z_mb
imputed_data <- generator %>% predict(list(as.matrix(X_mb), data_m), verbose=0)
```

```r
  # Combine observed and generated data for the final result
  imputed_data <- data_m * norm_data + (1 - data_m) * imputed_data

  # Renormalize the imputed data to the original scale and round if necessary
  imputed_data <- renormalization(imputed_data, norm_parameters) %>%
    rounding(data)

  imputed <- list(data = imputed_data,
                  D_loss = D_loss_values,
                  G_loss = G_loss_values
                  )
  # Return the imputed dataset
  return(imputed)
}
```

Implementation

In this section, we build two functions based on `gain`. The first one takes a complete dataset and assigns missing values randomly, while the second one takes two datasets that are actually the same, but one contains missing values, and then imputes the missing values and then compares the datasets to evaluate the imputation.

In both functions, we can see that a default version has been provided for all parameters. These parameters are those suggested by Yoon as the best for achieving the most performant GAIN possible.

gain_paper

This function, `gain_paper`, introduces missing values into a dataset and then applies the GAIN to impute the missing data. The function starts by determining the dimensions of the input dataset (number of rows and columns). It then creates a mask matrix, which identifies observed and missing values. The missing values are introduced randomly based on a specified **missing rate** (`miss_rate`). The resulting dataset, containing missing values, is then passed to the `gain` function for imputation.

Several hyperparameters influence the GAIN imputation process. `batch_size` defines how many samples are processed at a time, while `hint_rate` controls the probability of revealing hints about missing data to the discriminator. The generator loss is weighted using `alpha`, and optimization is performed using an Adam optimizer with learning rate (`learning`), momentum terms (`beta_1` and `beta_2`), and a chosen weight initialization strategy (`initialization`), among others. The function also provides options to monitor training loss (`loss_monitoring`), show a progress bar (`progress_bar`), and plot loss convergence over iterations (`plot_convergence`).

After training, the imputed dataset is compared with the original dataset using **RMSE**, but only for the artificially introduced missing values. The RMSE serves as a quantitative measure of imputation performance. If enabled, the function generates a loss convergence plot, visualizing how the generator and discriminator losses evolve over training iterations.

Finally, the function returns the **imputed dataset**, where missing values have been filled in using GAIN.

```r
# Function to introduce missing values and perform data imputation using GAIN
gain_paper <- function(data,
                       miss_rate = 0.2,   # Proportion of missing values to introduce
                       batch_size = 128,  # Number of samples per training batch
                       hint_rate = 0.9,   # Probability of hint matrix values being revealed
```

```r
                      alpha = 100,  # Weighting factor for generator loss
                      iterations = 10000,  # Number of training iterations
                      learning = 0.001,  # Learning rate for optimization
                      loss_monitoring = TRUE,  # Tracks loss values during training if enabled
                      beta_1 = 0.9,  # Beta1 parameter for Adam optimizer
                      beta_2 = 0.999,  # Beta2 parameter for Adam optimizer
                      initialization = "Xavier",  # Method for weight initialization
                      progress_bar = TRUE,  # Displays a progress bar if enabled
                      plot_convergence = TRUE) {  # Plots loss convergence if enabled

# Get the number of rows and columns in the input dataset
no <- nrow(data)  # Number of rows (observations)
dim <- ncol(data) # Number of columns (features)

# Create a mask matrix indicating observed (1) and missing (0) values
# `missig` generates a matrix where each value is 1 (observed) with probability (1 - miss_rate),
# and NA (missing) with probability `miss_rate`.
mask <- missig(1 - miss_rate, no, dim)

# Apply the mask to the original data to create a version with missing values
# Observed values remain intact, while missing entries are set to NA.
data_missing <- mask * data

# Perform imputation using the GAIN algorithm
imputed <- gain(data = data_missing,
                batch_size = batch_size,
                hint_rate = hint_rate,
                alpha = alpha,
                iterations = iterations,
                learning = learning,
                loss_monitoring = loss_monitoring,
                beta_1 = beta_1,
                beta_2 = beta_2,
                initialization = initialization,
                progress_bar = progress_bar)

# Compute the RMSE between the original and imputed data
# RMSE is calculated only for the missing entries.
rmse <- rmse_loss(data, imputed$data, mask)

# Print the RMSE to monitor imputation performance
cat("\n RMSE:", rmse)

# If enabled, plot the convergence of the GAN losses over iterations
if (plot_convergence == TRUE) {

  # Prepare data for plotting
  plot_data <- data.frame(
    iteration = 1:iterations,  # Sequence of iteration numbers
    G_loss = imputed$G_loss,  # Generator loss values
    D_loss = imputed$D_loss   # Discriminator loss values
  )
```

```
    # Generate the loss convergence plot
    plot <- ggplot(plot_data, aes(x = iteration)) +
      geom_line(aes(y = G_loss, color = "Generator Loss")) +  # Plot generator loss
      geom_line(aes(y = D_loss, color = "Discriminator Loss")) +  # Plot discriminator loss
      labs(
        title = "GAIN loss convergence",  # Title of the plot
        x = "Iterations",  # X-axis label
        y = "Loss",  # Y-axis label
        color = "Legend"  # Legend title
      ) +
      theme_minimal()

    # Display the plot
    print(plot)
  }

  # Return the final imputed dataset
  return(imputed$data)
}
```

gain_missing

This function, `gain_missing`, applies the GAIN algorithm to impute missing values in a dataset while also computing the imputation error against a reference **true dataset**.

The function takes two datasets as inputs: `data`, which contains missing values to be imputed, and `data_true`, which represents the original dataset before values were removed. The imputation process is handled by calling the `gain` function, which operates on `data` with several hyperparameters defining its training process, including `batch_size`, `hint_rate`, `alpha`, `iterations`, and optimizer settings (`learning`, `beta_1`, `beta_2`).

Before applying GAIN, a **mask matrix** is created using the `find_mask` function, which identifies which values are missing. The mask is essential for both guiding imputation and computing the **RMSE** of the imputation process. RMSE is computed only for the originally missing values, and then is printed for monitoring performance.

An optional feature of the function is the visualization of **loss convergence** over training iterations. If `plot_convergence` is set to `TRUE`, the function generates a line plot displaying the evolution of generator loss and discriminator loss. Finally, the function returns the **fully imputed dataset**, where missing values have been replaced with values estimated by GAIN.

```
gain_missing <- function(data, # dataset with missing values to be imputed
                         data_true, # original dataset
                         batch_size = 128,  # Number of samples per training batch
                         hint_rate = 0.9,  # Probability of hint matrix values being revealed
                         alpha = 100,  # Weighting factor for generator loss
                         iterations = 10000,  # Number of training iterations
                         learning = 0.001,  # Learning rate for optimization
                         loss_monitoring = TRUE,  # Tracks loss values during training if enabled
                         beta_1 = 0.9,  # Beta1 parameter for Adam optimizer
                         beta_2 = 0.999,  # Beta2 parameter for Adam optimizer
                         initialization = "Xavier",  # Method for weight initialization
                         progress_bar = TRUE,  # Displays a progress bar if enabled
```

```r
                      plot_convergence = TRUE) {  # Plots loss convergence if enabled

# Create mask for missing values
mask <- find_mask(data)

# Impute the missing values using the GAIN algorithm
# This uses the `gain` function with the specified parameters.
  imputed <- gain(data = data,
                  batch_size = batch_size,
                  hint_rate = hint_rate,
                  alpha = alpha,
                  iterations = iterations,
                  learning = learning,
                  loss_monitoring = loss_monitoring,
                  beta_1 = beta_1,
                  beta_2 = beta_2,
                  initialization = initialization,
                  progress_bar = progress_bar)

# Compute the RMSE between the original and imputed data
# RMSE is calculated only for the missing entries.
rmse <- rmse_loss(data_true, imputed$data, mask)

# Print the RMSE to monitor imputation performance
cat("\n RMSE:", rmse)

# If enabled, plot the convergence of the GAN losses over iterations
if (plot_convergence == TRUE) {

  # Prepare data for plotting
  plot_data <- data.frame(
    iteration = 1:iterations,  # Sequence of iteration numbers
    G_loss = imputed$G_loss,  # Generator loss values
    D_loss = imputed$D_loss   # Discriminator loss values
  )

  # Generate the loss convergence plot
  plot <- ggplot(plot_data, aes(x = iteration)) +
    geom_line(aes(y = G_loss, color = "Generator Loss")) +  # Plot generator loss
    geom_line(aes(y = D_loss, color = "Discriminator Loss")) +  # Plot discriminator loss
    labs(
      title = "GAIN loss convergence",  # Title of the plot
      x = "Iterations",  # X-axis label
      y = "Loss",  # Y-axis label
      color = "Legend"  # Legend title
    ) +
    theme_minimal()

  # Display the plot
  print(plot)
}

# Return the final imputed dataset
```

```r
  return(imputed$data)
}
```

Test

We now try to test GAIN on a dataset provided by Yoon and used in the original paper. The dataset in question is `letter`, consisting of 16 categorical variables and 20000 observations. We also download the `letter_miss` dataset, which is the `letter` dataset with 20% missing variables. We need this dataset in order to compare the results with the python implementation, as this will make the two software impute the same dataset.

```r
# Download the datasets
letter <- read_csv("letter.csv")
```

```
## Rows: 20000 Columns: 16
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (16): Cat 1, Cat 2, Cat 3, Cat 4, Cat 5, Cat 6, Cat 7, Cat 8, Cat 9, Cat...
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```r
letter_miss <- read_csv("letter_miss.csv")
```

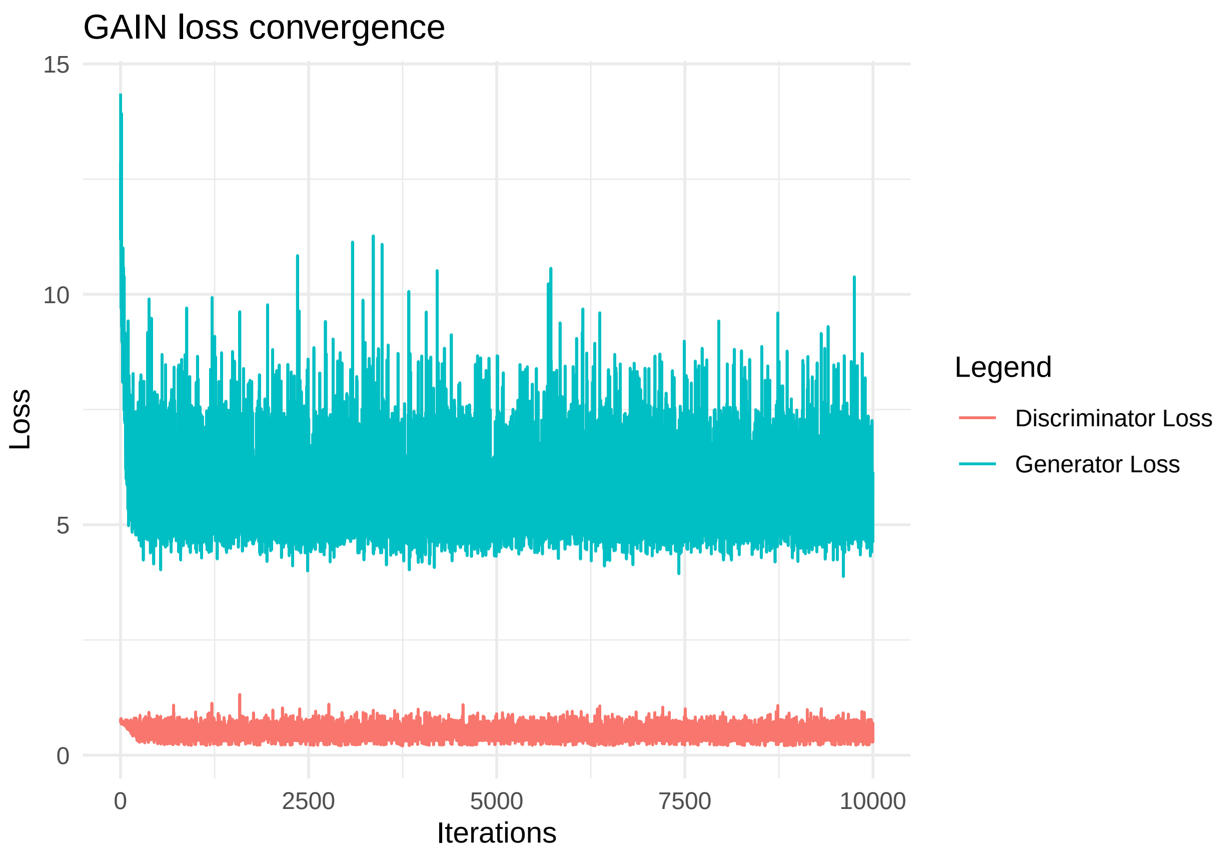
```
## Rows: 20000 Columns: 16
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (16): Cat 1, Cat 2, Cat 3, Cat 4, Cat 5, Cat 6, Cat 7, Cat 8, Cat 9, Cat...
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

We first test the GAIN with the `gain_paper` function. We obtain the graph of the convergence of the loss functions, which would seem to indicate that the functions converge much earlier than the 100000 interactions recommended by Yoon. We also obtain the RMSE which is equal to **0.1806**.

```r
# Set random seed for reproducibility
set.seed(130225)

# Impute missing values using the GAIN algorithm
imputed_paper <- gain_paper(letter)
```

```
## Iteration 100    Gloss 7.40426   Dloss 0.5962935Iteration 200    Gloss 5.130034  Dloss 0.535762Itera
##  RMSE: 0.1796347
```



This code performs data imputation using the GAIN algorithm and evaluates its performance by computing the Root Mean Squared Error (RMSE). The core imputation step is executed by calling `gain_missing(letter_miss, letter)`, where `letter_miss` represents the dataset with missing values, and `letter` is the original dataset. The function imputes missing values using GAIN, returning a completed dataset with estimated values.

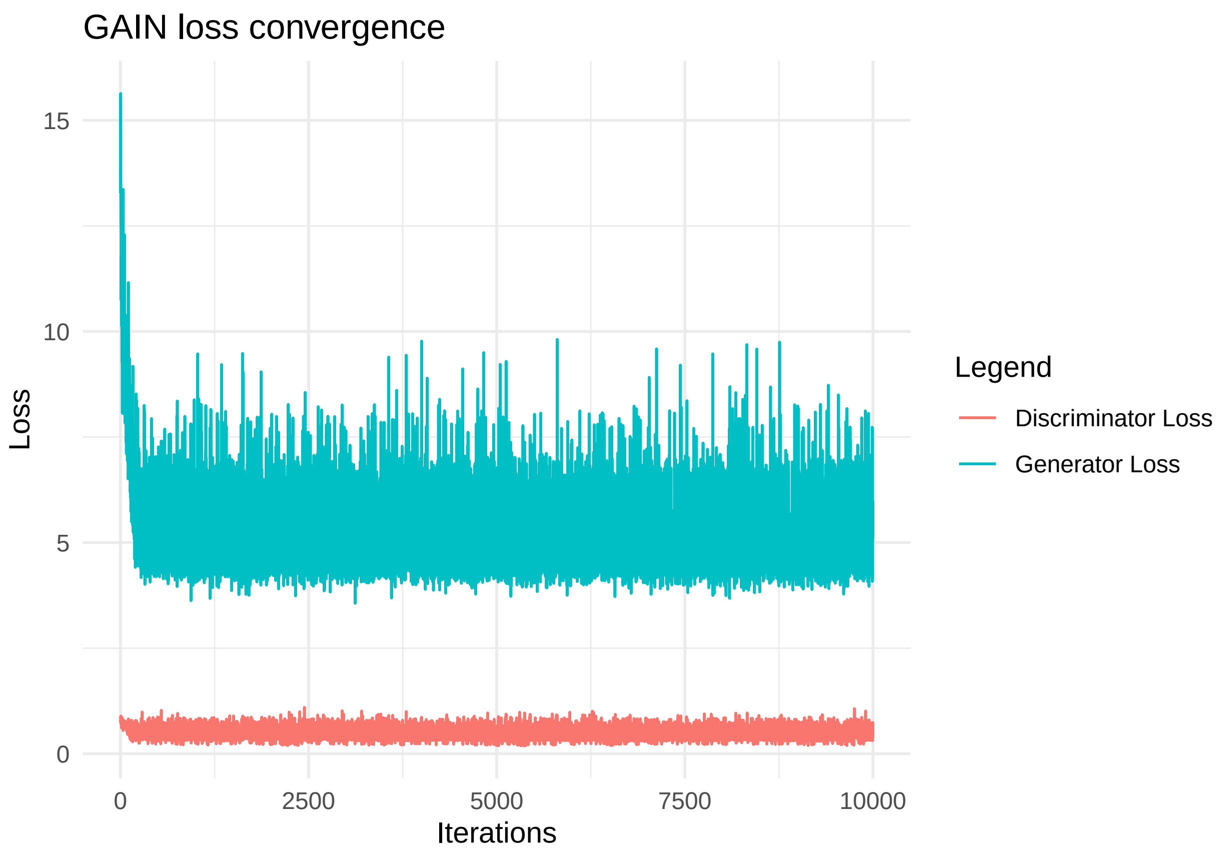
To compare different imputation approaches, an imputed dataset named `imputed_letter` is loaded from a CSV file (`imputed_letter_missing.csv`). This dataset has been pre-imputed using the original Python-based GAIN implementation by Yoon. We create the **mask matrix**, then the RMSE is computed using `rmse_loss`..

The RMSE for our R implementation is **0.1793**, which aligns with the previous since the dataset and missing percentage remain unchanged. However, when comparing with the Python version, a discrepancy appears: the Python-imputed dataset achieves an **RMSE of 0.1278**, significantly lower than our result. This suggests that the Python implementation might optimize better, indicating a potential difference in how the two versions function.

```r
# Set random seed for reproducibility
set.seed(130225)

# Impute missing values using the GAIN algorithm
imputed_missing <- gain_missing(letter_miss, letter)
```

```
## Iteration 100    Gloss 8.203135  Dloss 0.5794525Iteration 200    Gloss 7.110192  Dloss 0.732581Itera
##  RMSE: 0.1786429
```



```r
# Read the imputed data from a CSV file
imputed_letter <- read_csv("imputed_letter_missing.csv", col_names = F)
```

```
## Rows: 20000 Columns: 16
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (16): X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15, X16
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# Create a mask for the missing values in the original dataset
mask <- find_mask(letter_miss)

# Calculate the RMSE between the original and imputed datasets using the mask
rmse <- rmse_loss(letter, imputed_letter, mask)

# Print the RMSE value
cat("\n RMSE of the data imputed with Python:", rmse)
```

```
##
##  RMSE of the data imputed with Python: 0.1277718
```