Tuto - Beta-Optim

Jaad Belhouari - Fatima-Zahra Hannou - Aymane Mimoun

2025-03-28

Introduction

In this tutorial, we will guide you through building and training a machine learning model, quantifying uncertainties, and visualizing the results using the **`Algorithmique`** library in R.

We will use the **`xgboost`** model for training, along with methods for uncertainty quantification and coverage estimation. The example follows these steps:

1. Data Loading
2. Data Splitting
3. Model Building and Training
4. Uncertainty Quantification
5. Visualization of Results

1. Data Loading

First, we load the dataset and separate the features and labels.

```r
# Load necessary libraries
library(Algorithmique)
library(xgboost)
```

```
## Warning: le package 'xgboost' a été compilé avec la version R 4.3.3
```

```r
library(data.table)
```

```
## Warning: le package 'data.table' a été compilé avec la version R 4.3.3
```

```r
library(R6)
```

```
## Warning: le package 'R6' a été compilé avec la version R 4.3.2
```

```r
# Load dataset
setwd("../")
df <- fread("data/dataset.csv")
X <- df[, .(X1, X2)]
Y <- df[, !c("X1", "X2"), with = FALSE]

# Print dimensions of X and Y
print("The dimension of X is:")
```

```
## [1] "The dimension of X is:"
```

```r
print(dim(X))
```

```
## [1] 1000000       2
```

```r
print("The dimension of Y is:")
```

```
## [1] "The dimension of Y is:"
```

```r
print(dim(Y))
```

```
## [1] 1000000      24
```

2. Data Splitting

Now, we define the parameters for splitting the dataset into training, calibration, and test sets.

```r
# Define split parameters
n_train <- 10000
n_test <- 1000
n_calib <- 2000

# Perform data split
splits <- train_test_calib_split(X, Y, n_train = n_train, n_test = n_test, n_calib = n_calib)

# Extract split datasets
X_train <- splits$X_train
y_train <- splits$y_train
X_calib <- splits$X_calib
y_calib <- splits$y_calib
X_test <- splits$X_test
y_test <- splits$y_test
idx_calib <- splits$idx_calib
idx_test <- splits$idx_test
```

3. Model Building and Training

In this step, we create and train a model using xgboost.

```r
# Building our model
model <- MLModel$new(X_train, y_train, method = "gradient_boosting")
model$fit()
```

```r
# Ensure the "Saved_Models" directory exists
dir.create("Saved_Models", showWarnings = FALSE, recursive = TRUE)

# Save the model
saveRDS(model, file = "Saved_Models/MLModel_trained.rds")
```

4. Uncertainty Quantification

Now, we load the trained model and initialize the ModelUncertainties class to compute the uncertainty quantification using Beta quantiles.

```r
# Loading our pretrained model
loaded_model <- readRDS("Saved_Models/MLModel_trained.rds")
```

```r
# Initialize the ModelUncertainties class
uncertainty_model <- ModelUncertainties$new(
  model = loaded_model,
  X_calibration = X_calib,
  Y_calibration = y_calib,
  uncertainty_method = "Beta_Optim",
  Global_alpha = 0.1
)

# Fit the uncertainty method on the model (compute the Beta quantiles)
uncertainty_model$fit()
```

```
## Fitting uncertainty model with method: Beta_Optim
## Running dichotomy to find optimal Beta...
## We have 705 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.3525
## Beta = 0.0500 => Coverage = 0.3525
## Iteration 1: Beta = 0.0500, Empirical Simultaneous Coverage = 0.3525
## We have 1164 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.582
## Beta = 0.0250 => Coverage = 0.5820
## Iteration 2: Beta = 0.0250, Empirical Simultaneous Coverage = 0.5820
## We have 1527 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.7635
## Beta = 0.0125 => Coverage = 0.7635
## Iteration 3: Beta = 0.0125, Empirical Simultaneous Coverage = 0.7635
## We have 1759 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.8795
## Beta = 0.0062 => Coverage = 0.8795
## Iteration 4: Beta = 0.0062, Empirical Simultaneous Coverage = 0.8795
## We have 1892 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.946
## Beta = 0.0031 => Coverage = 0.9460
## Iteration 5: Beta = 0.0031, Empirical Simultaneous Coverage = 0.9460
## We have 1825 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.9125
## Beta = 0.0047 => Coverage = 0.9125
## Iteration 6: Beta = 0.0047, Empirical Simultaneous Coverage = 0.9125
## We have 1805 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.9025
## Beta = 0.0055 => Coverage = 0.9025
## Iteration 7: Beta = 0.0055, Empirical Simultaneous Coverage = 0.9025
## We have 1782 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.891
## Beta = 0.0059 => Coverage = 0.8910
## Iteration 8: Beta = 0.0059, Empirical Simultaneous Coverage = 0.8910
## We have 1782 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.891
## Beta = 0.0057 => Coverage = 0.8910
## Iteration 9: Beta = 0.0057, Empirical Simultaneous Coverage = 0.8910
## We have 1782 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.891
## Beta = 0.0056 => Coverage = 0.8910
## Iteration 10: Beta = 0.0056, Empirical Simultaneous Coverage = 0.8910
## We have 1805 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.9025
## Beta = 0.0055 => Coverage = 0.9025
## We have 1782 curves falling inside all their prediction intervals simultaneously
## For a total of 2000 curves
## Which leads to an empirical simultaneous coverage of 0.891
## Beta = 0.0056 => Coverage = 0.8910
## Optimal Beta found: 0.00546875
```

```r
# Check if our Beta value is bigger than Bonferroni one :
print(uncertainty_model$Beta_optim)
```

```
## [1] 0.00546875
```

```r
beta_bonferroni <- uncertainty_model$Global_alpha / ncol(y_test)
print(beta_bonferroni)
```

```
## [1] 0.004166667
```

```r
print("Beta Optim is bigger than Bonferroni one ?")
```

```
## [1] "Beta Optim is bigger than Bonferroni one ?"
```

```r
print(beta_bonferroni < uncertainty_model$Beta_optim)
```

```
## [1] TRUE
```

```r
# Get the prediction bounds
pred_bounds_test <- uncertainty_model$predict(X_test)

y_test_lower <- pred_bounds_test$y_lower
y_test_upper <- pred_bounds_test$y_upper
```

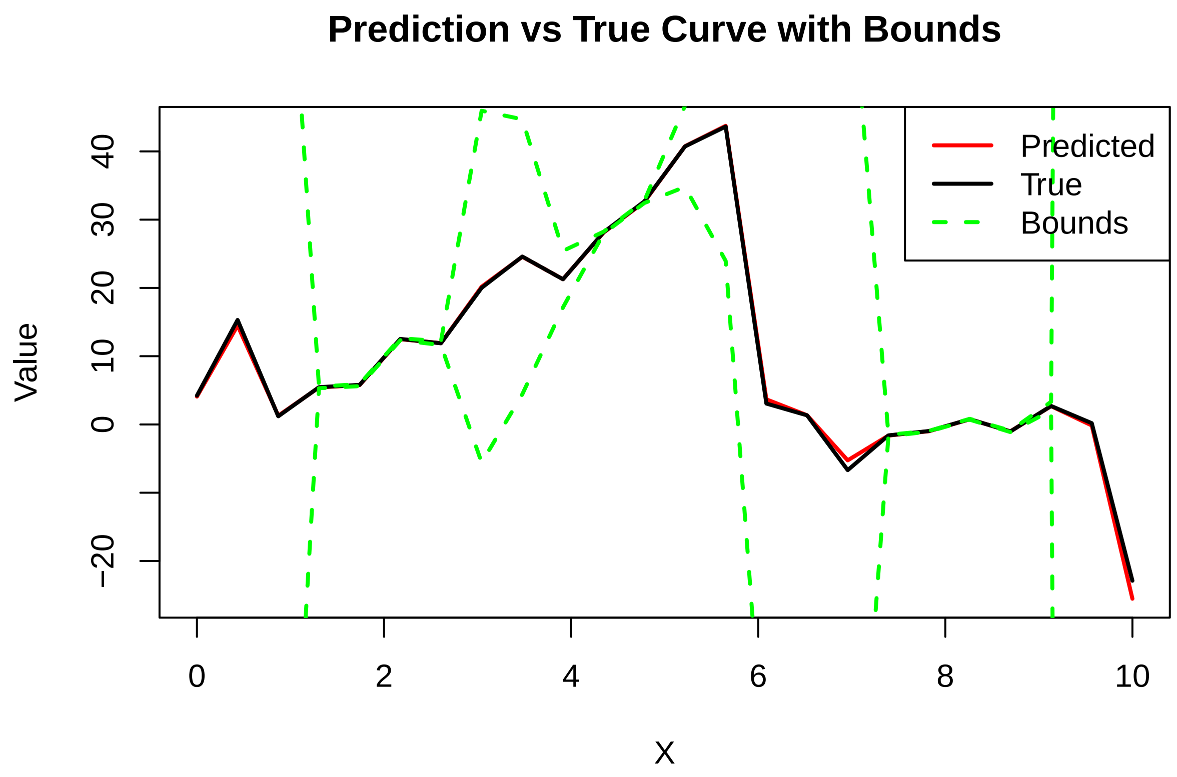
```r
# Compute the empirical coverage obtained of the test data

Empirical_simultaneous_coverage <- simultaneous_coverage(y_test, y_test_lower, y_test_upper)
```

```
## We have 864 curves falling inside all their prediction intervals simultaneously
## For a total of 1000 curves
## Which leads to an empirical simultaneous coverage of 0.864
```

5. Plot Visualization

Finally, we visualize the predicted curve, true values, and the upper and lower bounds for a particular test sample.

```r
# making prediction on the test set
y_pred_test <- loaded_model$predict(X_test)

# Define the x-axis values (24 points evenly spaced between 0 and 10)
x_values <- seq(0, 10, length.out = 24)

# Extract the relevant row (10th sample)
y_values_pred <- y_pred_test[10, ]  # Predicted curve (red)
y_values_true <- y_test[10, ]       # True curve (black)
y_up <- y_test_upper[10, ]     # Upper bound (green dashed)
y_low <- y_test_lower[10, ]     # Lower bound (green dashed)

# Plot predicted curve in red
plot(x_values, y_values_pred, type = "l", col = "red", lwd = 2,
     xlab = "X", ylab = "Value", main = "Prediction vs True Curve with Bounds")

# Add true curve in black
lines(x_values, y_values_true, col = "black", lwd = 2)

# Add upper and lower bounds in green (dashed)
lines(x_values, y_up, col = "green", lwd = 2, lty = 2)  # Dashed line
lines(x_values, y_low, col = "green", lwd = 2, lty = 2)  # Dashed line

# Add legend
legend("topright", legend = c("Predicted", "True", "Bounds"),
       col = c("red", "black", "green"), lwd = 2, lty = c(1, 1, 2))
```



6. Other uncertainty methods'

Our library provide 3 different uncertainty methods that can be apply to build the prediction intervals : Beta_Optim, Max_Rank and Max_Rank_Beta_Optim

6.1. Max Rank Method

```
# Initialize the ModelUncertainties class for a new method

choosen_method <- "Max_Rank"

uncertainty_model <- ModelUncertainties$new(
  model = loaded_model,
  X_calibration = X_calib,
  Y_calibration = y_calib,
  uncertainty_method = choosen_method,
  Global_alpha = 0.1
)

# Fit the uncertainty method on the model (compute the Beta quantiles)
uncertainty_model$fit()
```

```
## Fitting uncertainty model with method: Max_Rank
## Max Rank method fitted successfully. r_max = 1991
```

```r
# Get the prediction bounds
pred_bounds_test <- uncertainty_model$predict(X_test)

y_test_lower <- pred_bounds_test$y_lower
y_test_upper <- pred_bounds_test$y_upper
```

```r
# Compute the empirical coverage obtained of the test data

Empirical_simultaneous_coverage <- simultaneous_coverage(y_test, y_test_lower, y_test_upper)
```

```
## We have 864 curves falling inside all their prediction intervals simultaneously
## For a total of 1000 curves
## Which leads to an empirical simultaneous coverage of 0.864
```

6.2. Fast Beta Optim Method

```r
choosen_method <- "Fast_Beta_Optim"

uncertainty_model <- ModelUncertainties$new(
  model = loaded_model,
  X_calibration = X_calib,
  Y_calibration = y_calib,
  uncertainty_method = choosen_method,
  Global_alpha = 0.1
)

# Fit the uncertainty method on the model (compute the Beta quantiles)
uncertainty_model$fit()
```

```
## Fitting uncertainty model with method: Fast_Beta_Optim
## Running dichotomy to find optimal Beta...
## Beta = 0.0500 => Coverage = 0.3495
## Iteration 1: Beta = 0.0500, Empirical Simultaneous Coverage = 0.3495
## Beta = 0.0250 => Coverage = 0.5780
## Iteration 2: Beta = 0.0250, Empirical Simultaneous Coverage = 0.5780
## Beta = 0.0125 => Coverage = 0.7550
## Iteration 3: Beta = 0.0125, Empirical Simultaneous Coverage = 0.7550
## Beta = 0.0062 => Coverage = 0.8730
## Iteration 4: Beta = 0.0062, Empirical Simultaneous Coverage = 0.8730
## Beta = 0.0031 => Coverage = 0.9340
## Iteration 5: Beta = 0.0031, Empirical Simultaneous Coverage = 0.9340
## Beta = 0.0047 => Coverage = 0.9025
## Iteration 6: Beta = 0.0047, Empirical Simultaneous Coverage = 0.9025
## Beta = 0.0055 => Coverage = 0.8915
## Iteration 7: Beta = 0.0055, Empirical Simultaneous Coverage = 0.8915
## Beta = 0.0051 => Coverage = 0.8915
## Iteration 8: Beta = 0.0051, Empirical Simultaneous Coverage = 0.8915
## Beta = 0.0049 => Coverage = 0.9025
## Iteration 9: Beta = 0.0049, Empirical Simultaneous Coverage = 0.9025
## Beta = 0.0050 => Coverage = 0.9025
## Iteration 10: Beta = 0.0050, Empirical Simultaneous Coverage = 0.9025
## Beta = 0.0050 => Coverage = 0.9025
## Beta = 0.0051 => Coverage = 0.8915
## Optimal Beta found: 0.004980469
```

```
# Get the prediction bounds
pred_bounds_test <- uncertainty_model$predict(X_test)

y_test_lower <- pred_bounds_test$y_lower
y_test_upper <- pred_bounds_test$y_upper
```

```
# Compute the empirical coverage obtained of the test data

Empirical_simultaneous_coverage <- simultaneous_coverage(y_test, y_test_lower, y_test_upper)
```

```
## We have 876 curves falling inside all their prediction intervals simultaneously
## For a total of 1000 curves
## Which leads to an empirical simultaneous coverage of 0.876
```

7. Complexity Analysis

In this section, we analyze the computational complexity of the three uncertainty quantification methods (`Beta_Optim`, `Max_Rank`, and `Fast_Beta_Optim`) by varying the size of the calibration dataset and measuring the execution time for each method. We will perform the procedure 10 times for each calibration set size (`n_cal`), store the results, and plot the execution time versus `n_cal`.

7.1. Running and Measuring Execution Time

We define the range of calibration set sizes and then run the fitting process for each method, capturing the execution time. We repeat the procedure multiple times (10 times) to account for variability.

```
# Load the progress package
library(progress)
```

```
## Warning: le package 'progress' a été compilé avec la version R 4.3.3
```

```
# Define a function to measure execution time for different methods and calibration set sizes
measure_execution_time <- function(method, n_cal_values, model, X, Y, Global_alpha = 0.1, repetitions =

  # Store the execution times in a matrix
  execution_times <- matrix(NA, nrow = length(n_cal_values), ncol = repetitions)

  # Create a progress bar for the calibration sizes loop
  pb_cal <- progress_bar$new(
    format = "  Calibration Set Size [:bar] :percent Elapsed: :elapsedfull",
    total = length(n_cal_values), clear = TRUE, width = 60
  )

  # Loop over different n_cal sizes
  for (i in seq_along(n_cal_values)) {
```

```r
  n_cal <- n_cal_values[i]

  # Perform the data splitting (n_cal calibration set size)
  splits_list <- train_test_calib_split_for_complexity_analysis(X, Y, n_train = 10000, n_test = 1000,

  # Create a progress bar for repetitions loop
  pb_reps <- progress_bar$new(
    format = "    Repetition [:bar] :percent Elapsed: :elapsedfull",
    total = repetitions, clear = TRUE, width = 60
  )

  # Repeat the process 'repetitions' times to account for variability in execution time
  for (j in 1:repetitions) {
    # Extract the split data for the current draw
    splits <- splits_list[[j]]
    X_calib <- splits$X_calib
    y_calib <- splits$y_calib
    X_test <- splits$X_test

    # Initialize the ModelUncertainties class
    uncertainty_model <- ModelUncertainties$new(
      model = model,
      X_calibration = X_calib,
      Y_calibration = y_calib,
      uncertainty_method = method,
      Global_alpha = Global_alpha
    )

    # Suppress the verbose output by capturing it
    capture.output({
      # Measure the time taken to fit the model
      start_time <- Sys.time()
      uncertainty_model$fit()
      end_time <- Sys.time()
    }, file = NULL)  # Set file = NULL to discard the output

    # Store the execution time for this repetition
    execution_times[i, j] <- as.numeric(difftime(end_time, start_time, units = "secs"))

    # Update progress bar for repetitions
    pb_reps$tick()

    # Flush the console to ensure updates
    flush.console()
  }

  # Update the calibration set size progress bar
  pb_cal$tick()

  # Flush after each calibration loop to update the progress
  flush.console()
}
```

```r
  # Calculate the mean and variance (or standard deviation) for each n_cal value
  execution_stats <- data.frame(
    n_cal = n_cal_values,
    mean_time = apply(execution_times, 1, mean),
    variance_time = apply(execution_times, 1, var)  # You can replace var with sd for standard deviatio
  )

  return(execution_stats)
}
```

```r
# Define the range of n_cal values for the complexity analysis (logarithmic scale)
n_cal_values <- floor(5*10^seq(3, 5, length.out = 15))

# Measure the execution times for each uncertainty method and get the statistics
execution_stats_beta_optim <- measure_execution_time("Beta_Optim", n_cal_values, loaded_model, X, Y)
execution_stats_max_rank <- measure_execution_time("Max_Rank", n_cal_values, loaded_model, X, Y)
execution_stats_max_rank_beta_optim <- measure_execution_time("Fast_Beta_Optim", n_cal_values, loaded_mo
```

7.2. Plot Execution Time vs Calibration Set Size

Now, we can plot the execution time versus the size of the calibration set (n_cal) for each of the three uncertainty methods.

```r
# Load necessary library
library(ggplot2)
```

```
## Warning: le package 'ggplot2' a été compilé avec la version R 4.3.2
```

```r
library(dplyr)
```

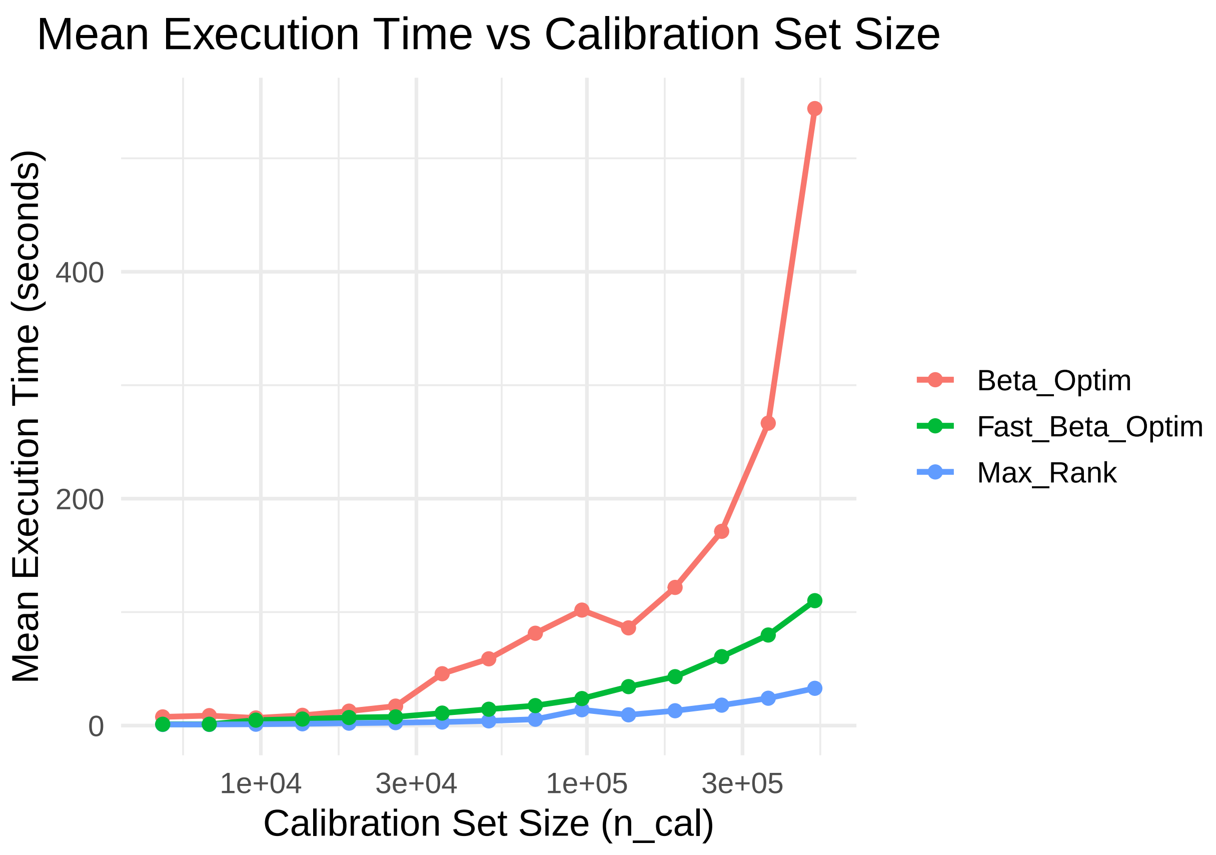
```
## Warning: le package 'dplyr' a été compilé avec la version R 4.3.2

##
## Attachement du package : 'dplyr'

## Les objets suivants sont masqués depuis 'package:data.table':
##
##     between, first, last

## L'objet suivant est masqué depuis 'package:xgboost':
##
##     slice

## Les objets suivants sont masqués depuis 'package:stats':
##
##     filter, lag

## Les objets suivants sont masqués depuis 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
# Combine the statistics from all methods into a single dataframe
execution_stats_df <- rbind(
  data.frame(method = "Beta_Optim", execution_stats_beta_optim),
  data.frame(method = "Max_Rank", execution_stats_max_rank),
  data.frame(method = "Fast_Beta_Optim", execution_stats_max_rank_beta_optim)
)
```

```r
# Compute standard deviation if not already available
execution_stats_df <- execution_stats_df %>%
  mutate(sd_time = sqrt(variance_time))

p <- ggplot(execution_stats_df, aes(x = n_cal, y = mean_time, color = method)) +
  geom_line(size = 1) +         # Épaisseur de la ligne augmentée
  geom_point(size = 2) +          # Taille des points augmentée
  geom_errorbar(aes(ymin = mean_time - sd_time, ymax = mean_time + sd_time),
              width = 0.05, size = 1) +  # Barres d'erreur plus épaisses
  scale_x_log10() +
  labs(
    title = "Mean Execution Time vs Calibration Set Size",
    x = "Calibration Set Size (n_cal)",
    y = "Mean Execution Time (seconds)"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    legend.title = element_blank(),
    plot.title = element_text(hjust = 0.5)
  )
```

```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

```r
# Display the plot
print(p)
```

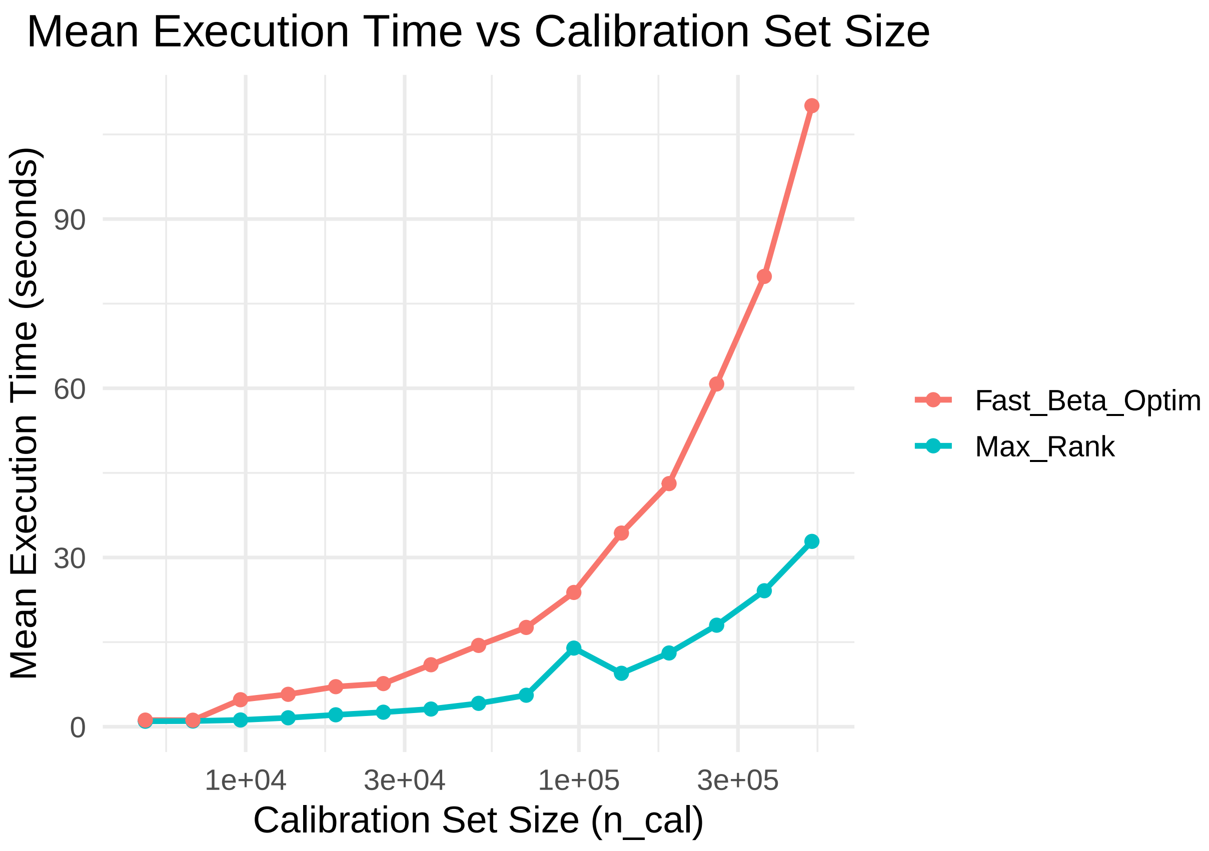


```r
# Save the plot with wider dimensions (e.g., 10x6 inches)
ggsave("mean_execution_time_with_errorbars.png", plot = p, width = 10, height = 6, dpi = 300)
```

Focus on Max Rank and Max Rank Beta Optim Methods:

```r
# Combine the statistics from all methods into a single dataframe
execution_stats_df <- rbind(
  #data.frame(method = "Beta_Optim", execution_stats_beta_optim),
  data.frame(method = "Max_Rank", execution_stats_max_rank),
  data.frame(method = "Fast_Beta_Optim", execution_stats_max_rank_beta_optim)
)
```

```r
# Compute standard deviation if not already available
execution_stats_df <- execution_stats_df %>%
  mutate(sd_time = sqrt(variance_time))

p <- ggplot(execution_stats_df, aes(x = n_cal, y = mean_time, color = method)) +
  geom_line(size = 1) +         # Épaisseur de la ligne augmentée
  geom_point(size = 2) +         # Taille des points augmentée
  geom_errorbar(aes(ymin = mean_time - sd_time, ymax = mean_time + sd_time),
                width = 0.05, size = 1) +  # Barres d'erreur plus épaisses
  scale_x_log10() +
  labs(
    title = "Mean Execution Time vs Calibration Set Size",
    x = "Calibration Set Size (n_cal)",
    y = "Mean Execution Time (seconds)"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    legend.title = element_blank(),
    plot.title = element_text(hjust = 0.5)
  )

# Display the plot
print(p)
```



```
# Save the plot with wider dimensions (e.g., 10x6 inches)
ggsave("mean_execution_time_with_errorbars_FastOptimVSMaxRank.png", plot = p, width = 10, height = 6, dp
```

7.3. Interpretation

In the resulting plot, you will see the execution times for each of the three methods plotted against the size of the calibration set (n_cal). This will help us understand how the computational complexity of each method scales with the size of the calibration set. Generally, you should expect methods that require sorting or rank-based operations (like Max_Rank and Max_Rank_Beta_Optim) to have higher computational complexity compared to methods that use direct optimization (like Beta_Optim).

The log scale for the x-axis allows us to better visualize the differences in execution times across a wide range of calibration set sizes, especially when the size increases significantly.

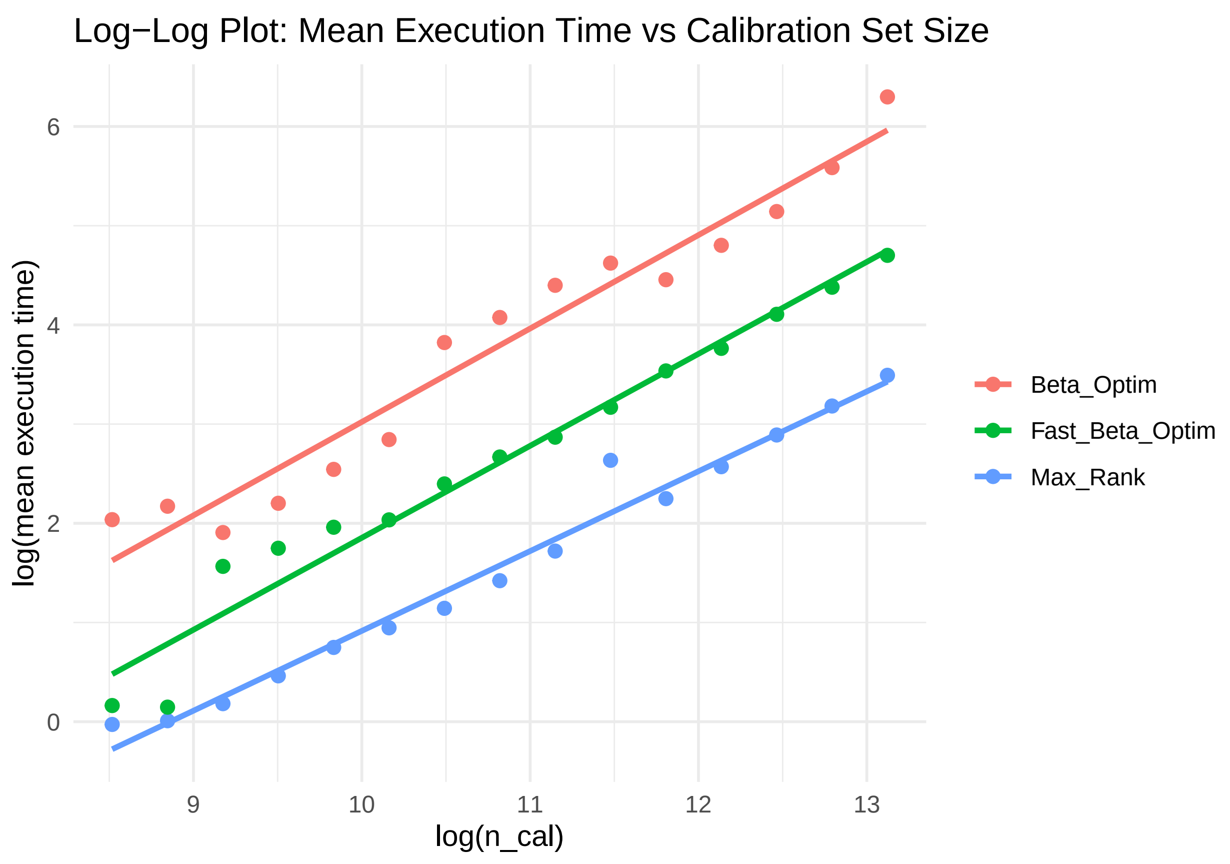
7.4. Coefficient of the exponent

Now, let's try to get the coefficient of the exponent for the execution time of each method.

```r
# Combine the statistics from all methods into a single dataframe
execution_stats_df <- rbind(
  data.frame(method = "Beta_Optim", execution_stats_beta_optim),
  data.frame(method = "Max_Rank", execution_stats_max_rank),
  data.frame(method = "Fast_Beta_Optim", execution_stats_max_rank_beta_optim)
)
```

```r
# Take the log of calibration size and mean execution time
execution_stats_df <- execution_stats_df %>%
  mutate(
    log_n_cal = log(n_cal),
    log_mean_time = log(mean_time)
  )

# Plot log-log for visual inspection
p <- ggplot(execution_stats_df, aes(x = log_n_cal, y = log_mean_time, color = method)) +
  geom_point(size = 2) +
  geom_smooth(method = "lm", se = FALSE, size = 1) +
  labs(
    title = "Log-Log Plot: Mean Execution Time vs Calibration Set Size",
    x = "log(n_cal)",
    y = "log(mean execution time)"
  ) +
  theme_minimal() +
  theme(legend.title = element_blank())

print(p)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



```
# Save the plot with wider dimensions (e.g., 10x6 inches)
ggsave("loglogplot.png", plot = p, width = 10, height = 6, dpi = 300)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

```
# Compute the exponent k for each method
library(broom)
```

```
## Warning: le package 'broom' a été compilé avec la version R 4.3.2
```

```
exponent_estimates <- execution_stats_df %>%
  group_by(method) %>%
  do(tidy(lm(log_mean_time ~ log_n_cal, data = .))) %>%
  filter(term == "log_n_cal") %>%
  select(method, estimate) %>%
  rename(exponent_k = estimate)

print(exponent_estimates)
```

```
## # A tibble: 3 x 2
## # Groups:   method [3]
##   method          exponent_k
##   <chr>                <dbl>
## 1 Beta_Optim           0.942
## 2 Fast_Beta_Optim      0.927
## 3 Max_Rank             0.805
```

New try without the point evaluated with n__cal = 2000

Indeed, having exponent values below 1 isn't normal as we should have a nlog(n) complexity. This issue araise because the execution time computed for n__cal = 2000 distorpes the slope and induice false value for k.

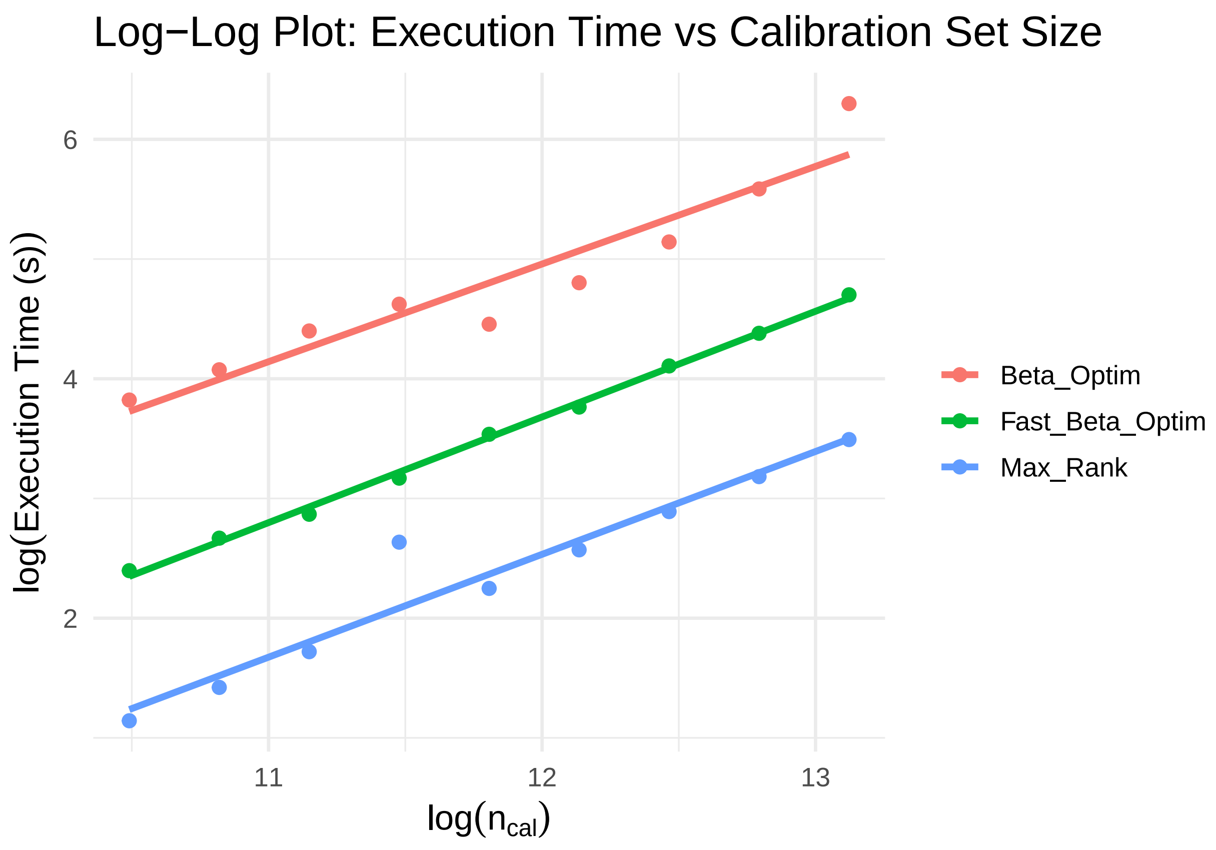
```r
colnames(execution_stats_df)
```

```
## [1] "method"        "n_cal"         "mean_time"     "variance_time"
## [5] "log_n_cal"     "log_mean_time"
```

```r
library(dplyr)

# Get the 6 smallest n_cal values
n_cal_to_exclude <- sort(unique(execution_stats_df$n_cal))[1:6]

# Filter each method by excluding those 6 smallest values
execution_stats_beta_optim_filtered <- execution_stats_df %>%
  filter(method == "Beta_Optim", !n_cal %in% n_cal_to_exclude)

execution_stats_max_rank_filtered <- execution_stats_df %>%
  filter(method == "Max_Rank", !n_cal %in% n_cal_to_exclude)

execution_stats_max_rank_beta_optim_filtered <- execution_stats_df %>%
  filter(method == "Fast_Beta_Optim", !n_cal %in% n_cal_to_exclude)
```

```r
# --- Step 1: Combine data ---
execution_stats_df <- rbind(
  data.frame(method = "Beta_Optim", execution_stats_beta_optim_filtered),
  data.frame(method = "Max_Rank", execution_stats_max_rank_filtered),
  data.frame(method = "Fast_Beta_Optim", execution_stats_max_rank_beta_optim_filtered)
)

# --- Step 2: Clean & log-transform ---
execution_stats_df <- execution_stats_df %>%
  filter(mean_time > 0, n_cal > 0) %>%  # remove zeroes if any
  mutate(
    log_n_cal = log(n_cal),
    log_mean_time = log(mean_time)
  )

# --- Step 3: Visual Log-Log Plot (include all points for plot) ---
ggplot(execution_stats_df, aes(x = log_n_cal, y = log_mean_time, color = method)) +
  geom_point(size = 2) +
  geom_smooth(method = "lm", se = FALSE, size = 1.2) +
  labs(
    title = "Log-Log Plot: Execution Time vs Calibration Set Size",
    x = expression(log(n[cal])),
    y = expression(log("Execution Time (s)"))
  ) +
  theme_minimal(base_size = 13) +
  theme(legend.title = element_blank())
```

```
## 'geom_smooth()' using formula = 'y ~ x'
```



```r
# --- Step 4: Compute the exponent k (remove small calibration sizes from fitting) ---
library(broom)

# You can change this threshold depending on your use case
min_cal_threshold <- 50000

exponent_estimates <- execution_stats_df %>%
  filter(n_cal >= min_cal_threshold) %>%  # <--- filter out small sizes here
  group_by(method) %>%
  do(tidy(lm(log_mean_time ~ log_n_cal, data = .))) %>%
  filter(term == "log_n_cal") %>%
  select(method, exponent_k = estimate)

print(exponent_estimates)
```

```
## # A tibble: 3 x 2
## # Groups:   method [3]
##   method          exponent_k
##   <chr>                <dbl>
## 1 Beta_Optim           0.847
## 2 Fast_Beta_Optim      0.899
## 3 Max_Rank             0.829
```

8. Conclusion

This analysis gives us insight into how each uncertainty quantification method behaves with respect to the size of the calibration data, and it helps us choose the best method depending on the computational resources available and the desired accuracy of the uncertainty estimates.