R Notebook

```
#clear global environment
rm(list=ls())
```

```
#read in data
data<-read.csv("/Users/Fionn/Downloads/framingham.csv")
head(data)
#check which variables have missing values
colnames(data)[colSums(is.na(data)) > 0]
```

```
#for Polya-Gamma Augmentation
library("BayesLogit")
#for data cleaning
library("mice")
#for data manipulation
library("dplyr")
```

```
# Convert categorical variables(that have NA's) to factors
data <- data %>%
  mutate(
    education = factor(education),
    BPMeds = factor(BPMeds)
  )

# Verify structure
str(data)
```

```
imputation_methods <- make.method(data)
 # Predictive Mean Matching for numeric variables
imputation_methods[c("totChol", "BMI", "heartRate", "glucose", "cigsPerDay")] <- "pmm"
imputation_methods["education"] <- "polr"  # ordered probit
imputation_methods["BPMeds"] <- "logreg"   # Logistic regression for binary categorical
# Run multiple imputation
imputed_data <- mice(data, method = imputation_methods, m = 5, seed = 123)

# Inspect imputed values
#print(imputed_data)
# Select one imputed dataset
filled_data <- complete(imputed_data, 1)
```

```
#check no NA's remain
sum(is.na(filled_data))
str(filled_data$education)
```

```r
#need to create dummy variables in education for logistic regression
#make 1st level of education a reference category
one_hot_encode <- function(df, cat_var) {
  df[[cat_var]] <- droplevels(factor(df[[cat_var]]))  # Ensure it's a factor
  dummies <- model.matrix(~ df[[cat_var]], data = df)[, -1, drop = FALSE]  # Remove first column (refer
  colnames(dummies) <- sub("df\\[\\[cat_var\\]\\]", cat_var, colnames(dummies))  #  # Find the original
  col_index <- which(names(df) == cat_var)

  # Create a new dataframe with the dummy columns inserted at the original position
  df_new <- cbind(df[ , 1:(col_index - 1), drop = FALSE],  # Columns before categorical
                  dummies,  # One-hot encoded variables
                  df[ , (col_index + 1):ncol(df), drop = FALSE])  # Columns after

  return(df_new)
}
final.data <- one_hot_encode(filled_data, "education")
head(final.data)
```

```r
# Convert categorical variables to numeric
data_numeric <- final.data%>%
  mutate(across(where(is.character), as.factor)) %>%  # Convert character to factor
  mutate(across(where(is.factor), as.numeric)) %>%  # Convert factor to numeric
  mutate(intercept = 1)  # Add intercept before converting to matrix

# Define outcome and predictor variables
y <- data_numeric$TenYearCHD  # Outcome column
X <- data_numeric %>%
  dplyr::select(-TenYearCHD) %>%  # Predictor columns
  as.matrix()  # Convert to matrix

# Check structure
print(str(X))  # Ensure X is numeric
print(dim(X))  # Confirm correct dimensions
print(head(X))  # Preview the first few rows
```

```r
#set seed before generating random indices
set.seed(2003)
#hold back some rows to make predictions at
testindices <- sample(c(1:4240), size=100)
trainindices <- c(1:4240)[-testindices]
#partition test data
test.data <- X[testindices , ,drop=F]
test.resp <- y[testindices]
#partition training data
train.data <- X[trainindices , ,drop=F]
train.resp <- y[trainindices]
```

```r
# function to set up Gibbs Sampler for beta using Polya-Gamma augmentation
gibbs_sampler <- function(y, X, n_iter = 5000, b_prior = NULL, B_prior_inv = NULL) {

  # Data dimensions
  N <- length(y)  # Number of observations
  P <- ncol(X)    # Number of predictors (including intercept)
```

```
  # Set priors if not provided
  if (is.null(b_prior)) {
    b_prior <- rep(0, P)  # Mean of Gaussian prior
  }
  if (is.null(B_prior_inv)) {
    B_prior_inv <- diag(1, P)  # Precision matrix
  }

  # Initialize storage for beta samples
  beta_samples <- matrix(0, nrow = n_iter, ncol = P)

  # Initial values
  set.seed(123)
  beta_init<-rnorm(P, mean=0, sd=0.01)
  beta <- beta_init  # Start with values close to zero

  # Gibbs sampling loop
  for (iter in 1:n_iter) {

    # Compute psi = X * beta (log-odds)
    psi <- X %*% beta

    # Sample omega from Polya-Gamma distribution
    omega <- rpg(N, rep(1, N), psi)  # PG(1, psi) for logistic regression

    # Construct diagonal matrix Omega
   #Omega <- diag(as.vector(omega), N, N)

    # Compute kappa
    kappa <- y - 0.5  # Equivalent to (y - n/2) since n = 1

    # Compute posterior covariance and mean
    # t(X) %*% (X * omega)
    A       <- t(X) %*%(X*omega)+ B_prior_inv
    R      <- chol(A)
    R.inv   <-backsolve(R, diag(rep(1,ncol(R))))
    V_omega <- R.inv%*%t(R.inv)
    m_omega <- V_omega %*% (t(X) %*% kappa + B_prior_inv %*% b_prior)

    # Generate N(0,1) deviates in length of m_omega
    z_i <- rnorm(length(m_omega), 0, 1)
    #apply transformation so beta~MVN(m_omega,V_omega)
    beta <- (R.inv%*%z_i)+m_omega

    # Store the sample
    beta_samples[iter, ] <- beta
  }

  return(beta_samples)
}
```

```
#Generate samples of beta on training data, take prior to be N(0,1000)
beta_samples <- gibbs_sampler(y=train.resp , X=train.data, n_iter = 5000,b_prior=rep(0, 18), B_prior_inv
```

```r
#Preview means of beta
beta_posterior_mean <- colMeans(beta_samples)
# Print results
print(beta_posterior_mean)
```

```r
# quick gut check to see that estimates obtained from Polya Gamma are reasonable
  glm_fit <- glm(train.resp ~ train.data - 1, family = binomial)  #remove intercept from X

  # Extract MLE coefficients
  glm_coefs <- coef(glm_fit)
 glm_coefs
   #should be close to b-hat-posterior mean
```

```r
#Compute Model Accuracy and confusion matrix
# Compute linear predictor (log-odds)
log_odds <- train.data %*% beta_posterior_mean

# Convert to probabilities using the sigmoid function
predicted_probs <- 1 / (1 + exp(-log_odds))
predicted_labels <- ifelse(predicted_probs > 0.5, 1, 0)
# Compute Accuracy
accuracy <- mean(predicted_labels == train.resp)
print(paste("Model Accuracy:", round(accuracy, 3)))

# Create Confusion Matrix
table(Predicted = predicted_labels, Actual = train.resp)
```

```r
#may be redundant.,.,..
#find laplace approximation to the posterior
source("/Users/Fionn/Downloads/ModelEvidenceLogistic.R")
eobj <- evidence.obj( y, X, rep(0, 18) ,diag(0.01, 18) )

bhat <- eobj$newton.method()
regfisherinf <- -eobj$hessian( bhat )
postprec <- chol2inv( chol(regfisherinf))
gl <- glm( y~X-1, family=binomial)
# check
gl$coefficients # should be close to bhat
bhat
```

```r
#function to do AOI importance sampling to estimate the posterior predictive
logistic_posterior_predictive <- function(Y, X, y, x_new, theta) {
  # Y: binary outcome (0,1) for existing observation
  # X: design vector for existing observation (including intercept)
  # y: reference outcome value (0,1) for computing weights
  # x_new: design vector for new point (for computing weights)
  # theta: MCMC parameter samples (each column is a sample)

  n_samples <- ncol(theta)

  # Define dispatch table with just two cases (y=0 and y=1)
  dispatch_table <- list(
```

```
    # Case: y is 1
    "one" = function() {
      # Compute logistic probabilities for y=1 at x_new
      w <- sapply(1:n_samples, function(t) {
        1 / (1 + exp(-x_new %*% theta[, t]))
      })

      # Normalize weights
      normalized_weights <- w/sum(w)

      # Compute weighted likelihoods
      weighted_likelihoods <- sapply(1:n_samples, function(t) {
        normalized_weights[t] * (1 / (1 + exp(-X %*% theta[, t])))
      })

      if(Y == 1) {
        return(sum(weighted_likelihoods))
      } else { # Y == 0
        return(1 - sum(weighted_likelihoods))
      }
    },

    # Case: y is 0
    "zero" = function() {
      # Compute logistic probabilities for y=1 at x_new
      w <- sapply(1:n_samples, function(t) {
        1 / (1 + exp(-x_new %*% theta[, t]))
      })

      # Normalize weights for y=0
      normalized_weights <- (1-w)/sum(1-w)

      # Compute weighted likelihoods
      weighted_likelihoods <- sapply(1:n_samples, function(t) {
        normalized_weights[t] * (1 / (1 + exp(X %*% theta[, t])))
      })

      if(Y == 1) {
        return(sum(weighted_likelihoods))
      } else { # Y == 0
        return(1 - sum(weighted_likelihoods))
      }
    }
  )

  # Execute appropriate function based on y value
  if(y == 1) {
    return(dispatch_table[["one"]]())
  } else { # y == 0
    return(dispatch_table[["zero"]]())
  }
}
```

```r
logistic_posterior_predictive_vec <- function(Y, X, y, x_new, theta) {
  # Y: vector of binary outcomes (0,1) for existing observations
  # X: design matrix where each row corresponds to an observation in Y
  # y: reference outcome value (0,1) for computing weights
  # x_new: design vector for new point (for computing weights)
  # theta: MCMC parameter samples (each column is a sample)

  n_samples <- ncol(theta)
  n_obs <- length(Y)

  # Calculate x_new probabilities once for all samples
  log_probs_new <- drop(1 / (1 + exp(-x_new %*% theta)))

  # Pre-allocate matrix for all probabilities
  all_probs <- matrix(0, nrow=n_obs, ncol=n_samples)

  if(y == 1) {
    # Normalize weights for y=1
    normalized_weights <- log_probs_new / sum(log_probs_new)

    # Compute probabilities for each observation and sample
    for(i in 1:n_obs) {
      all_probs[i,] <- drop(1 / (1 + exp(-X[i,,drop=FALSE] %*% theta)))
    }

  } else { # y == 0
    # Normalize weights for y=0
    normalized_weights <- (1 - log_probs_new) / sum(1 - log_probs_new)

    # Compute probabilities for each observation and sample
    for(i in 1:n_obs) {
      all_probs[i,] <- drop(1 / (1 + exp(X[i,,drop=FALSE] %*% theta)))
    }
  }

  # Apply weights to each column (each sample)
  weighted_probs <- sweep(all_probs, 2, normalized_weights, "*")

  # Sum across samples for each observation
  weighted_sums <- rowSums(weighted_probs)

  # Final result based on Y values
  results <- ifelse(Y == 1, weighted_sums, 1 - weighted_sums)

  return(results)
}
```

```r
#debugging

#logistic_posterior_predictive_vec(test.resp[1:10],test.data[c(1:10), , drop=F], y=0, x_new=x_new[i, ,
#logistic_posterior_predictive(test.resp[1],test.data[1, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
#logistic_posterior_predictive(test.resp[2],test.data[2, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
#logistic_posterior_predictive(test.resp[3],test.data[3, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
```

```r
#logistic_posterior_predictive(test.resp[4],test.data[4, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
#logistic_posterior_predictive(test.resp[5],test.data[5, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
#logistic_posterior_predictive(test.resp[6],test.data[6, , drop=F], y=0, x_new=x_new[i, , drop = FALSE]
```

```r
#modify function to output string containing prediction set
full_conformal_classify <- function(Y, X, y_grid, x_new, theta, alpha) {
  # Track which values are accepted
  accepted_0 <- FALSE
  accepted_1 <- FALSE

  for (l in 1:length(y_grid)) {
    # Conformity scores on dataset
    sig_1_to_n <- logistic_posterior_predictive_vec(Y = Y, X = X, y = y_grid[[l]], x_new = x_new, theta

    # Conformity score on test point
    sig_n_plus_one <- logistic_posterior_predictive(Y = y_grid[[l]], X = x_new, y = y_grid[[l]], x_new =
    # Adjusted quantile calculation
    n <- length(Y)
    pi <- (length(which(sig_1_to_n <= sig_n_plus_one)) + 1) / (n + 1)
    # Reject points if pi <= alpha
    if (pi > alpha) {
      # Mark the value as accepted
      if (y_grid[[l]] == 0) {
        accepted_0 <- TRUE
      } else if (y_grid[[l]] == 1) {
        accepted_1 <- TRUE
      }
    }
  }

  # Format the output string based on which values were accepted
  if (accepted_0 && accepted_1) {
    return("{0,1}")
  } else if (accepted_0) {
    return("{0}")
  } else if (accepted_1) {
    return("{1}")
  } else {
    return("{}")
  }
}
```

```r
#transpose beta for input
beta<-t(beta_samples)
#create exact Y grid
y.grid<-list(0,1)
#assign held back points to x_new
x_new <- test.data
dim(x_new)
```

```r
# Preallocate store for efficiency
store <- vector("character", length = nrow(x_new))
```

```r
# Loop through each row of x_new
for (i in 1:nrow(x_new)) {
  store[i] <- full_conformal_classify(
    Y = train.resp,
    X = train.data,
    y_grid = y.grid,
    x_new = x_new[i, , drop = FALSE],  # Iterate through rows
    theta = beta,
    alpha = 0.15
  )

  # Progress indicator every 10 iterations
  if (i %% 10 == 0 || i == nrow(x_new)) {
    cat(sprintf("Progress: %d/%d iterations completed.\n", i, nrow(x_new)))
    flush.console()  # Ensure output appears in real-time
  }
}
```

```r
store
test.resp
```

```r
length(which(store=="{}" | store== "{0,1}"))
```

```r
bayes.pred <- function(x_new, theta, alpha) {
  # Efficiently compute all likelihoods in one operation
  # For each row in x_new, compute probability for each column in theta
  n_rows <- nrow(x_new)
  n_cols <- ncol(theta)

  # Pre-allocate matrix to store all probabilities
  all_probs <- matrix(0, nrow=n_rows, ncol=n_cols)

  # Compute probabilities for all observations and all theta samples
  for(i in 1:n_rows) {
    linear_pred <- x_new[i,] %*% theta
    all_probs[i,] <- 1 / (1 + exp(-linear_pred))
  }

  # Calculate mean probability for each row
  p_means <- rowMeans(all_probs)

  # Create result vector
  results <- vector("character", n_rows)

  # Assign prediction sets based on conditions
  results[1-p_means >= 1-alpha] <- "{0}"
  results[p_means >= 1-alpha] <- "{1}"
  results[pmax(1-p_means, p_means) <= 1-alpha] <- "{0,1}"

  return(results)
}
```

```r
bayes.pred(x_new,beta,0.1)
which(bayes.pred(x_new,beta,0.1)=="{1}")
which(test.resp==1)
```

```r
length(which(bayes.pred(x_new,beta,0.14)=="{0,1}"))
```

```r
compute_single_element_misclassification <- function(predictions, actuals) {
  # Input validation
  if(length(predictions) != length(actuals)) {
    stop("Predictions and actuals must have the same length")
  }

  n <- length(predictions)

  # Initialize counters
  single_element_count <- 0
  incorrect_count <- 0

  for(i in 1:n) {
    pred <- predictions[i]
    actual <- actuals[i]

    # Only consider single element predictions
    if(pred == "{0}" || pred == "{1}") {
      single_element_count <- single_element_count + 1

      # Check if prediction is correct
      if((pred == "{0}" && actual != 0) || (pred == "{1}" && actual != 1)) {
        incorrect_count <- incorrect_count + 1
      }
    }
  }

  # Calculate misclassification rate for single element predictions
  misclassification_rate <- if(single_element_count > 0) {
    incorrect_count / single_element_count
  } else {
    NA  # No single element predictions to evaluate
  }

  return(list(
    single_element_predictions = single_element_count,
    incorrect_predictions = incorrect_count,
    misclassification_rate = misclassification_rate,
    single_element_proportion = single_element_count / n
  ))
}
```

```r
compute_single_element_misclassification(predictions = store, actuals=test.resp)
compute_single_element_misclassification(predictions = bayes.pred(x_new,beta,0.15), actuals=test.resp)
```

```r
plot_uninformative_rates <- function(train.resp, train.data, y.grid, x_new, theta,
                                     alpha_values = seq(0.01, 0.5, by = 0.05)) {

  # Create storage for results
  results <- data.frame(
    alpha = alpha_values,
    empty_count = 0,
    full_count = 0,
    uninformative_rate = 0
  )

  # For each alpha value
  for (a_idx in 1:length(alpha_values)) {
    alpha <- alpha_values[a_idx]
    cat(sprintf("Processing alpha = %.2f (%d/%d)\n", alpha, a_idx, length(alpha_values)))

    # Preallocate store for efficiency
    store <- vector("character", length = nrow(x_new))

    # Loop through each row of x_new
    for (i in 1:nrow(x_new)) {
      store[i] <- full_conformal_classify(
        Y = train.resp,
        X = train.data,
        y_grid = y.grid,
        x_new = x_new[i, , drop = FALSE],
        theta = theta,
        alpha = alpha
      )

      # Optional progress indicator
      if (i %% 50 == 0 || i == nrow(x_new)) {
        cat(sprintf("  Progress: %d/%d rows completed\r", i, nrow(x_new)))
        flush.console()
      }
    }

    # Count empty and full sets
    empty_count <- sum(store == "{}")
    full_count <- sum(store == "{0,1}")
    uninformative_count <- empty_count + full_count

    # Store results
    results$empty_count[a_idx] <- empty_count
    results$full_count[a_idx] <- full_count
    results$uninformative_rate[a_idx] <- uninformative_count/nrow(x_new)

    cat("\n")
  }

  # Plot the results
  par(mfrow=c(1,2))
```

```
  # Plot uninformative rate
  plot(results$alpha, results$uninformative_rate, type="o", col="blue",
       xlab="Alpha", ylab="Uninformative Prediction Rate",
       main="Rate of Uninformative Predictions")
  grid()

  # Plot breakdown of empty vs full sets
  barplot(t(as.matrix(results[,c("empty_count", "full_count")])),
          beside=TRUE, names.arg=results$alpha,
          col=c("red", "green"),
          main="Breakdown of Uninformative Predictions",
          xlab="Alpha", ylab="Count")
  legend("topright", legend=c("Empty sets", "Full sets {0,1}"),
         fill=c("red", "green"))

  # Reset plot settings
  par(mfrow=c(1,1))

  return(results)
}
```

```
# Define alpha values to test
alpha_values <- seq(0.1, 0.2, by = 0.01)

# Run analysis
results <- plot_uninformative_rates(
  train.resp = train.resp,
  train.data = train.data,
  y.grid = y.grid,
  x_new = x_new,
  theta = beta,
  alpha_values = alpha_values
)
```

```
# Print results table
print(results)
```

```
bayes_plot_uninformative_rates <- function(x_new, theta,
                                    alpha_values = seq(0.01, 0.5, by = 0.05)) {

  # Create storage for results
  results <- data.frame(
    alpha = alpha_values,
    empty_count = 0,
    full_count = 0,
    uninformative_rate = 0
  )

  # For each alpha value
  for (a_idx in 1:length(alpha_values)) {
    alpha <- alpha_values[a_idx]
    cat(sprintf("Processing alpha = %.2f (%d/%d)\n", alpha, a_idx, length(alpha_values)))
```

```
    # Preallocate store for efficiency
    store <- vector("character", length = nrow(x_new))

    # Loop through each row of x_new
    for (i in 1:nrow(x_new)) {
      # Pass only the current row to bayes.pred
      store[i] <- bayes.pred(x_new = x_new[i, , drop=FALSE], theta = theta, alpha = alpha)

      # Optional progress indicator
      if (i %% 50 == 0 || i == nrow(x_new)) {
        cat(sprintf("  Progress: %d/%d rows completed\r", i, nrow(x_new)))
        flush.console()
      }
    }

    # Count empty and full sets
    empty_count <- sum(store == "{}")
    full_count <- sum(store == "{0,1}")
    uninformative_count <- empty_count + full_count

    # Store results
    results$empty_count[a_idx] <- empty_count
    results$full_count[a_idx] <- full_count
    results$uninformative_rate[a_idx] <- uninformative_count/nrow(x_new)

    cat("\n")
  }

  # Plot the results
  par(mfrow=c(1,2))

  # Plot uninformative rate
  plot(results$alpha, results$uninformative_rate, type="o", col="blue",
       xlab="Alpha", ylab="Uninformative Prediction Rate",
       main="Rate of Uninformative Predictions")
  grid()

  # Plot breakdown of empty vs full sets
  barplot(t(as.matrix(results[,c("empty_count", "full_count")])),
          beside=TRUE, names.arg=results$alpha,
          col=c("red", "green"),
          main="Breakdown of Uninformative Predictions",
          xlab="Alpha", ylab="Count")
  legend("topright", legend=c( "Both"),
         fill=c("green"))

  # Reset plot settings
  par(mfrow=c(1,1))

  return(results)
}
```

```r
# Define alpha values to test
alpha_values <- seq(0.1, 0.5, by = 0.05)

# Run analysis
results <- bayes_plot_uninformative_rates(
  x_new = x_new,
  theta = beta,
  alpha_values = alpha_values
)
```

```r
# Print results table
print(results)
```

```r
bayes_plot_missclassify_rates <- function(x_new, y_true, theta,
                                  alpha_values = seq(0.01, 0.5, by = 0.05)) {

  # Create storage for results
  results <- data.frame(
    alpha = alpha_values,
    empty_count = 0,
    full_count = 0,
    single_element_count = 0,
    misclassification_count = 0,
    uninformative_rate = 0,
    misclassification_rate = 0
  )

  # For each alpha value
  for (a_idx in 1:length(alpha_values)) {
    alpha <- alpha_values[a_idx]
    cat(sprintf("Processing alpha = %.2f (%d/%d)\n", alpha, a_idx, length(alpha_values)))

    # Preallocate store for efficiency
    store <- vector("character", length = nrow(x_new))

    # Loop through each row of x_new
    for (i in 1:nrow(x_new)) {
      # Pass only the current row to bayes.pred
      store[i] <- bayes.pred(x_new = x_new[i, , drop=FALSE], theta = theta, alpha = alpha)

      # Optional progress indicator
      if (i %% 50 == 0 || i == nrow(x_new)) {
        cat(sprintf("  Progress: %d/%d rows completed\r", i, nrow(x_new)))
        flush.console()
      }
    }

    # Count empty and full sets
    empty_count <- sum(store == "{}")
    full_count <- sum(store == "{0,1}")
    uninformative_count <- empty_count + full_count

    # Count single-element predictions and misclassifications
```

```
    single_0_indices <- store == "{0}"
    single_1_indices <- store == "{1}"
    single_element_count <- sum(single_0_indices) + sum(single_1_indices)

    # Calculate misclassifications for single-element predictions
    misclassifications <- 0

    # For predictions of {0}, check if true label is 1
    misclassifications <- misclassifications + sum(single_0_indices & y_true == 1)

    # For predictions of {1}, check if true label is 0
    misclassifications <- misclassifications + sum(single_1_indices & y_true == 0)

    # Store results
    results$empty_count[a_idx] <- empty_count
    results$full_count[a_idx] <- full_count
    results$single_element_count[a_idx] <- single_element_count
    results$misclassification_count[a_idx] <- misclassifications
    results$uninformative_rate[a_idx] <- uninformative_count/nrow(x_new)

    # Calculate misclassification rate (only among single-element predictions)
    if (single_element_count > 0) {
      results$misclassification_rate[a_idx] <- misclassifications/single_element_count
    } else {
      results$misclassification_rate[a_idx] <- NA
    }

    cat("\n")
  }

  # Plot only the misclassification rate
  plot(results$alpha, results$misclassification_rate, type="o", col="purple",
       xlab="Alpha", ylab="Misclassification Rate",
       main="Misclassification Rate (Single-Element Predictions)")
  grid()

  return(results)
}
```

```
bayes_plot_missclassify_rates(x_new= x_new, y_true=test.resp, theta=beta,alpha_values = seq(0.1, 0.5, by
```

```
conf_plot_missclassify_rates <- function(train.resp, train.data, y.grid, x_new, y_true, theta,
                                 alpha_values = seq(0.01, 0.5, by = 0.05)) {

  # Create storage for results
  results <- data.frame(
    alpha = alpha_values,
    empty_count = 0,
    full_count = 0,
    single_element_count = 0,
    misclassification_count = 0,
    uninformative_rate = 0,
    misclassification_rate = 0
```

```
)

# For each alpha value
for (a_idx in 1:length(alpha_values)) {
  alpha <- alpha_values[a_idx]
  cat(sprintf("Processing alpha = %.2f (%d/%d)\n", alpha, a_idx, length(alpha_values)))

  # Preallocate store for efficiency
  store <- vector("character", length = nrow(x_new))

  # Loop through each row of x_new
  for (i in 1:nrow(x_new)) {
    # Pass only the current row to full_conformal_classify
    store[i] <- full_conformal_classify(
      Y = train.resp,
      X = train.data,
      y_grid = y.grid,
      x_new = x_new[i, , drop = FALSE],
      theta = theta,
      alpha = alpha
    )

    # Optional progress indicator
    if (i %% 50 == 0 || i == nrow(x_new)) {
      cat(sprintf("  Progress: %d/%d rows completed\r", i, nrow(x_new)))
      flush.console()
    }
  }

  # Count empty and full sets
  empty_count <- sum(store == "{}")
  full_count <- sum(store == "{0,1}")
  uninformative_count <- empty_count + full_count

  # Count single-element predictions and misclassifications
  single_0_indices <- store == "{0}"
  single_1_indices <- store == "{1}"
  single_element_count <- sum(single_0_indices) + sum(single_1_indices)

  # Calculate misclassifications for single-element predictions
  misclassifications <- 0

  # For predictions of {0}, check if true label is 1
  misclassifications <- misclassifications + sum(single_0_indices & y_true == 1)

  # For predictions of {1}, check if true label is 0
  misclassifications <- misclassifications + sum(single_1_indices & y_true == 0)

  # Store results
  results$empty_count[a_idx] <- empty_count
  results$full_count[a_idx] <- full_count
  results$single_element_count[a_idx] <- single_element_count
  results$misclassification_count[a_idx] <- misclassifications
```

```
    results$uninformative_rate[a_idx] <- uninformative_count/nrow(x_new)

    # Calculate misclassification rate (only among single-element predictions)
    if (single_element_count > 0) {
      results$misclassification_rate[a_idx] <- misclassifications/single_element_count
    } else {
      results$misclassification_rate[a_idx] <- NA
    }

    cat("\n")
  }

  # Plot only the misclassification rate
  plot(results$alpha, results$misclassification_rate, type="o", col="purple",
       xlab="Alpha", ylab="Misclassification Rate",
       main="Misclassification Rate (Single-Element Predictions)")
  grid()

  return(results)
}
```

```
conf_plot_missclassify_rates(train.resp, train.data, y.grid, x_new, y_true=test.resp, theta=beta,
                                  alpha_values = seq(0.1, 0.5, by = 0.05))
```