Final Project

STAT 117

Simon Ma

May 9, 2025

SECTION 1: INTRODUCTION

In this project, I develop three classification models, namely logistic regression, random forest, and gradient boosting machines (GBM), to predict binary pathological complete response (pCR) based on gene expression data. To mimic real-world deployment scenarios, such as a hospital making predictions for a new cohort of patients, we designate studies 4 through 16 (excluding study 10) as the training set and hold out study 10 for final testing. All internal model validation is performed using a leave-one-study-out (LOSO) cross-validation strategy to assess generalizability across datasets.

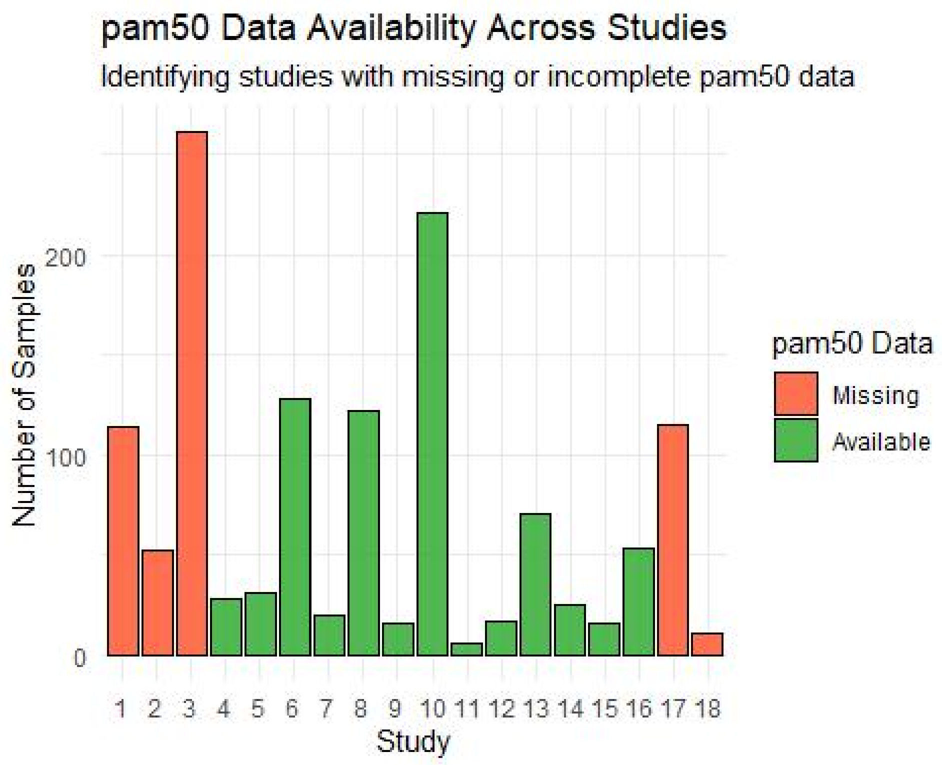
The modeling pipeline combines rigorous feature engineering with both interpretable and flexible learning methods. After computing AUC scores for each gene and filtering by threshold, selected genes are sign-adjusted and grouped based on absolute pairwise correlation. Within each group, we compute weighted averages to form aggregate predictors, which are then compressed using Principal Component Analysis (PCA). These engineered features are used to train all three classifiers. Logistic regression is chosen as a baseline due to its simplicity and interpretability in clinical contexts, while random forest and GBM are included as nonlinear ensemble methods capable of capturing complex interactions.

All three models outperform a benchmark logistic regression model trained directly on the PAM50 gene panel, in both training and test AUC. Notably, the random forest model achieves the highest test AUC (0.754), indicating strong generalization capability.

First it is necessary to obtain an initial visualization of the data without any cleaning.

```r
load("C:/Users/lordg/Downloads/pCR_binary.RData")

# Verification of pam50 data availability across all studies
cat("Analyzing pam50 data availability across all studies:\n\n")
```

```
## Analyzing pam50 data availability across all studies:

# Create a summary table for all studies
study_summary <- data.frame(
  Study = integer(),
  TotalSamples = integer(),
  NACount = integer(),
  NAPercentage = numeric(),
  HasPam50Data = logical(),
  stringsAsFactors = FALSE
)
# Analyze each study
for (i in 1:length(pam50_pCR)) {
  if (!is.null(pam50_pCR[[i]])) {
    total_samples <- length(pam50_pCR[[i]])
    na_count <- sum(is.na(pam50_pCR[[i]]))
    na_percentage <- round(100 * na_count / total_samples, 2)
    has_pam50 <- na_count < total_samples  # At least some non-NA value
s

    # Add to summary table
    study_summary <- rbind(study_summary, data.frame(
      Study = i,
      TotalSamples = total_samples,
      NACount = na_count,
      NAPercentage = na_percentage,
      HasPam50Data = has_pam50
    ))

    cat(sprintf("Study %d: %d samples, %d NAs (%.2f%%), Has pam50 data:
 %s\n",
                i, total_samples, na_count, na_percentage, has_pam50))
  } else {
    # Element is NULL
    study_summary <- rbind(study_summary, data.frame(
      Study = i,
      TotalSamples = 0,
      NACount = 0,
      NAPercentage = 100,
      HasPam50Data = FALSE
    ))

    cat(sprintf("Study %d: pam50_pCR is NULL\n", i))
  }
}

## Study 1: 114 samples, 114 NAs (100.00%), Has pam50 data: FALSE
## Study 2: 53 samples, 53 NAs (100.00%), Has pam50 data: FALSE
## Study 3: 261 samples, 261 NAs (100.00%), Has pam50 data: FALSE
## Study 4: 28 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 5: 31 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 6: 128 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 7: 20 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 8: 122 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 9: 16 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 10: 221 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 11: 6 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 12: 17 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 13: 71 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 14: 25 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 15: 16 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 16: 54 samples, 0 NAs (0.00%), Has pam50 data: TRUE
## Study 17: 115 samples, 115 NAs (100.00%), Has pam50 data: FALSE
## Study 18: 11 samples, 11 NAs (100.00%), Has pam50 data: FALSE

# Create a visualization of pam50 data availability
library(ggplot2)

# Include all studies for the visualization
all_studies <- data.frame()
for (i in 1:length(pam50_pCR)) {
  if (!is.null(pam50_pCR[[i]])) {
    total_samples <- length(pam50_pCR[[i]])
    na_count <- sum(is.na(pam50_pCR[[i]]))
    non_na_count <- total_samples - na_count

    all_studies <- rbind(all_studies, data.frame(
      Study = i,
      Available = non_na_count,
      Missing = na_count
    ))
  } else {
    all_studies <- rbind(all_studies, data.frame(
      Study = i,
      Available = 0,
      Missing = 0
    ))
  }
}

# Reshape data for plotting
all_studies_melted <- melt(all_studies, id.vars = "Study")

# Create the plot
ggplot(all_studies_melted, aes(
  fill = factor(variable, levels = c("Missing", "Available")),
  y = value,
  x = as.factor(Study)
)) +
  geom_bar(position = "stack", stat = "identity", color = "black", alph
a = 0.85) +
  labs(
    title = "pam50 Data Availability Across Studies",
    subtitle = "Identifying studies with missing or incomplete pam50 da
ta",
    x = "Study",
    y = "Number of Samples",
    fill = "pam50 Data"
  ) +
  theme_minimal() +
  scale_fill_manual(values = c(
    "Missing" = "#FF5733",
    "Available" = "#33AA33"
  ))
```

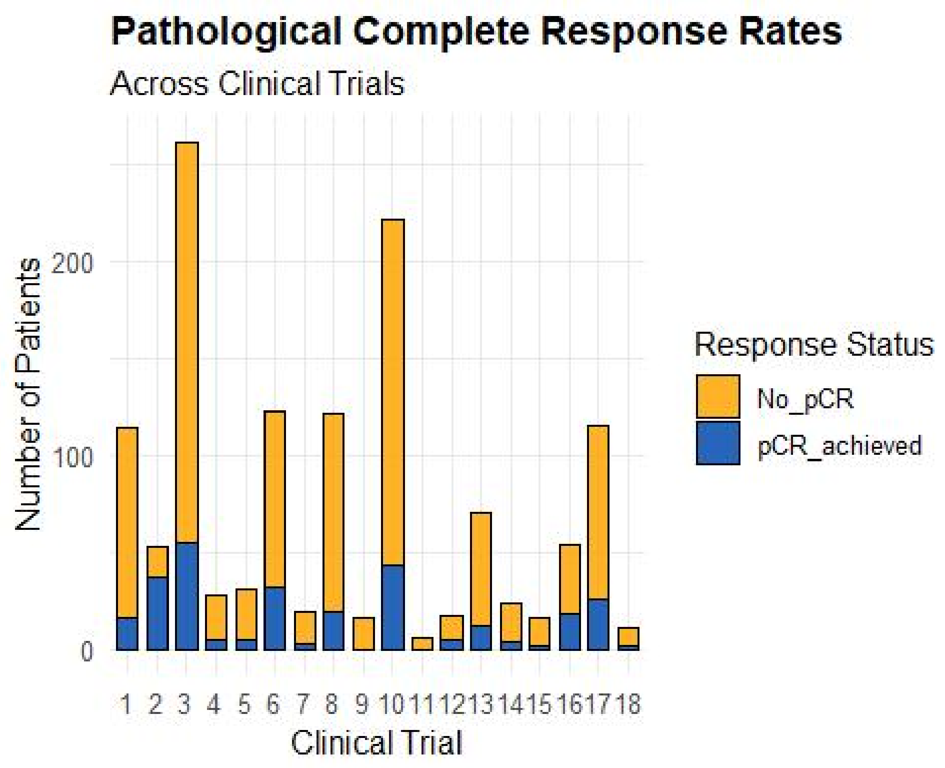


```
# Identify studies without pam50 data
studies_without_pam50 <- study_summary$Study[!study_summary$HasPam50Dat
a]
cat("\nStudies without pam50 data:", paste(studies_without_pam50, colla
pse = ", "), "\n")

##
## Studies without pam50 data: 1, 2, 3, 17, 18

# Identify studies with partial pam50 data (some NAs)
studies_with_partial_pam50 <- study_summary$Study[
  study_summary$HasPam50Data & study_summary$NAPercentage > 0
]
cat("Studies with incomplete pam50 data:",
    ifelse(length(studies_with_partial_pam50) > 0,
           paste(studies_with_partial_pam50, collapse = ", "),
           "None"), "\n")

## Studies with incomplete pam50 data: None

# Identify studies with complete pam50 data
studies_with_complete_pam50 <- study_summary$Study[
  study_summary$HasPam50Data & study_summary$NAPercentage == 0
]
cat("Studies with complete pam50 data:",
    paste(studies_with_complete_pam50, collapse = ", "), "\n")

## Studies with complete pam50 data: 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16

# Recommendation based on findings
cat("\nRecommendation: Based on the analysis, studies",
    paste(studies_without_pam50, collapse = ", "),
    "should be excluded from further analysis as they lack pam50 data r
equired for comparison with benchmark models.\n")

##
## Recommendation: Based on the analysis, studies 1, 2, 3, 17, 18 shoul
d be excluded from further analysis as they lack pam50 data required fo
r comparison with benchmark models.
```

Secondly, to get a better train-test split and study the class balances and quantity of datapoints in the pCR response variable, we plot a bar chart of pCR counts by study.

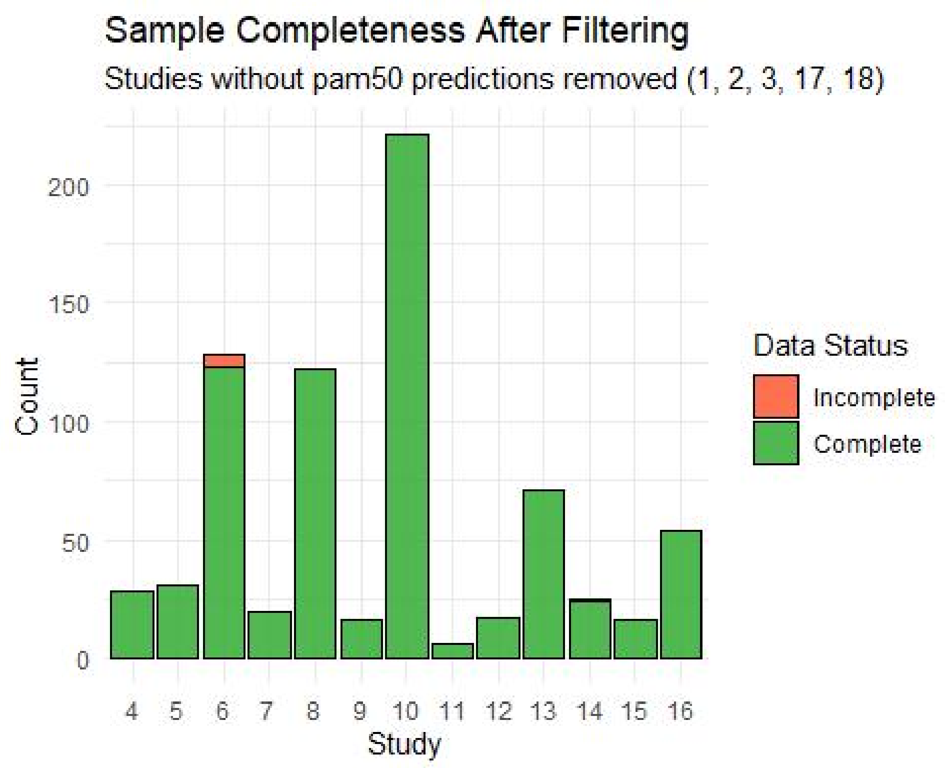
```
# Stacked bar plot of pathological response by trial (without ESR1)

# Initialize the response data
response_data <- cbind(
  pCR_achieved   = sum(pCR[[1]][!is.na(pCR[[1]])] == 1),
  No_pCR         = sum(pCR[[1]][!is.na(pCR[[1]])] == 0),
  trial          = 1
)

# Aggregate pCR data for each trial
for (i in 2:length(XX_pCR)) {
  response_data <- rbind(
    response_data,
    cbind(
      pCR_achieved   = sum(pCR[[i]][!is.na(pCR[[i]])] == 1),
      No_pCR         = sum(pCR[[i]][!is.na(pCR[[i]])] == 0),
      trial          = i
    )
```

```r
  )
}

# Convert response data into a dataframe
response_data <- as.data.frame(response_data)

# Reshape the data for ggplot
response_data <- melt(response_data, id.vars = "trial")

# Create the stacked bar plot
ggplot(response_data, aes(
  fill = factor(variable, levels = c("No_pCR", "pCR_achieved")),
  y = value,
  x = as.factor(trial)
)) +
  geom_bar(position = "stack", stat = "identity",
           color = "black", alpha = 0.85, width = 0.7) +
  labs(
    title    = "Pathological Complete Response Rates",
    subtitle = "Across Clinical Trials",
    x        = "Clinical Trial",
    y        = "Number of Patients",
    fill     = "Response Status"
  ) +
  theme_minimal(base_size = 12) +
  theme(plot.title = element_text(face = "bold")) +
  scale_fill_manual(values = c(
    "No_pCR" = "#FFA500",
    "pCR_achieved" = "#004AAD"
  ))
```

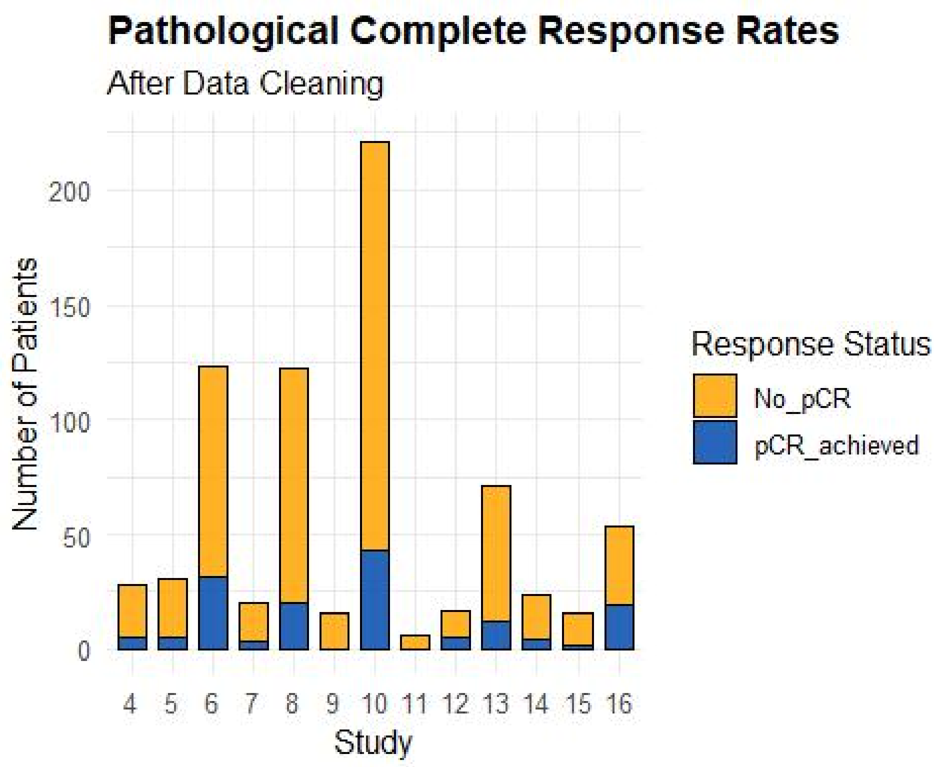


The bar chart reveals that some cohorts (Studies 9 and 11) contribute only incomplete pCR labels, and many other studies also exhibit substantial imbalance between responders and non-responders. This precludes using raw accuracy as a meaningful metric. Instead, we focus on AUC for ranking performance.

```r
# 1. Remove studies without pam50 predictions (1, 2, 3, 17, and 18)
studies_to_keep <- setdiff(1:18, c(1, 2, 3, 17, 18))
pCR_filtered <- pCR[studies_to_keep]
XX_pCR_filtered <- XX_pCR[studies_to_keep]
pam50_pCR_filtered <- pam50_pCR[studies_to_keep]

# 2. Handle missing data in each study
# - Drop rows with NA values for pCR

# Track complete and incomplete counts for each study
complete_counts <- numeric(length(studies_to_keep))
incomplete_counts <- numeric(length(studies_to_keep))
study_indices <- numeric(length(studies_to_keep))

for (i in 1:length(studies_to_keep)) {
  study_idx <- studies_to_keep[i]
  study_indices[i] <- study_idx

  # Count NAs in pCR before filtering
  total_samples <- length(pCR_filtered[[i]])
  na_pCR_count <- sum(is.na(pCR_filtered[[i]]))

  # Filter out NA pCR values
  non_na_pCR <- !is.na(pCR_filtered[[i]])
  pCR_filtered[[i]] <- pCR_filtered[[i]][non_na_pCR]
  XX_pCR_filtered[[i]] <- XX_pCR_filtered[[i]][, non_na_pCR, drop = FAL
SE]
  pam50_pCR_filtered[[i]] <- pam50_pCR_filtered[[i]][non_na_pCR]

  # Count complete vs incomplete after filtering
  complete_counts[i] <- length(pCR_filtered[[i]])
  incomplete_counts[i] <- total_samples - complete_counts[i]
}

# Create a data frame for visualization
study_data <- data.frame(
  study = study_indices,
  Complete = complete_counts,
  Incomplete = incomplete_counts
)

# Reshape for plotting
study_data_melted <- melt(study_data, id.vars = "study")

# Create a bar chart showing complete vs incomplete samples
ggplot(study_data_melted, aes(
  fill = factor(variable, levels = c("Incomplete", "Complete")),
  y = value,
  x = as.factor(study)
)) +
  geom_bar(position = "stack", stat = "identity", color = "black", alph
a = 0.85) +
  labs(
    title = "Sample Completeness After Filtering",
    subtitle = "Studies without pam50 predictions removed (1, 2, 3, 17,
 18)",
    x = "Study",
    y = "Count",
    fill = "Data Status"
  ) +
  theme_minimal() +
  scale_fill_manual(values = c(
    "Incomplete" = "#FF5733",
    "Complete" = "#33AA33"
  ))
```



```
# Calculate and visualize pCR distribution in cleaned data
pCR_stats <- data.frame(
  study = study_indices,
  pCR_achieved = sapply(pCR_filtered, function(x) sum(x == 1)),
  No_pCR = sapply(pCR_filtered, function(x) sum(x == 0))
)

# Reshape for plotting
pCR_stats_melted <- melt(pCR_stats, id.vars = "study")

# Create a bar chart showing pCR distribution
ggplot(pCR_stats_melted, aes(
  fill = factor(variable, levels = c("No_pCR", "pCR_achieved")),
  y = value,
  x = as.factor(study)
)) +
  geom_bar(position = "stack", stat = "identity", color = "black", alph
a = 0.85, width = 0.7) +
  labs(
    title = "Pathological Complete Response Rates",
    subtitle = "After Data Cleaning",
    x = "Study",
    y = "Number of Patients",
    fill = "Response Status"
  ) +
  theme_minimal(base_size = 12) +
  theme(plot.title = element_text(face = "bold")) +
  scale_fill_manual(values = c(
    "No_pCR" = "#FFA500",
    "pCR_achieved" = "#004AAD"
  ))
```



```
# Summary statistics
cat("Total samples after filtering:", sum(complete_counts), "\n")

## Total samples after filtering: 749

cat("Total pCR achieved:", sum(sapply(pCR_filtered, function(x) sum(x =
= 1))), "\n")

## Total pCR achieved: 150

cat("Total no pCR:", sum(sapply(pCR_filtered, function(x) sum(x == 0))),
 "\n")

## Total no pCR: 599

cat("Overall pCR rate:", round(100 * sum(sapply(pCR_filtered, function
(x) sum(x == 1))) /
                          sum(complete_counts), 2), "%\n")

## Overall pCR rate: 20.03 %

# Return cleaned datasets for further analysis
# pCR_filtered, XX_pCR_filtered, pam50_pCR_filtered
```

To handle missing data, we drop any samples with missing pCR values and impute missing gene expression values with 0. This simple strategy avoids more complex methods like semi-supervised learning or advanced imputation, which we deemed less impactful than decisions around feature selection and model design.

SECTION 2: CLEANING, STANDARDIZATION

Before modeling, each study’s expression matrix will be standardized—first centering and scaling within-study, then applying quantile normalization across the training set—to align feature distributions and mitigate inter-study technical variability. Together, these steps ensure that our pipelines reflect realistic deployment (predicting on a different and new hospital) and deliver reliable, well-calibrated risk estimates even when responder rates are low.

Standardization procedure

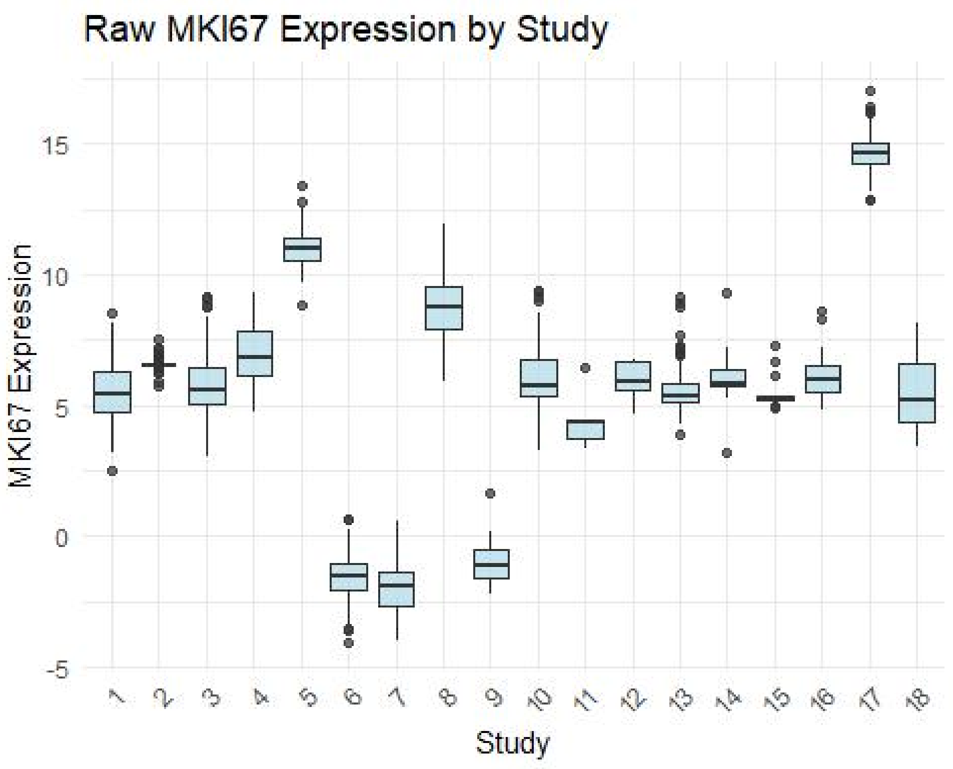
We begin by ensuring that all studies contain the same set of genes. This is achieved by identifying the common genes across all datasets, which guarantees that we are comparing the same set of biomarkers across the studies. Once the common genes are identified, each study is filtered to only include these shared genes, ensuring that the subsequent analysis is consistent.

Next, we correct for batch effects by normalizing the gene expression data relative to the expression of MKI67, a proliferation marker that is assumed to have consistent expression patterns across studies. This correction involves adjusting the gene expression data by subtracting the MKI67 expression and then scaling it by the standard deviation of MKI67 expression within each study. This ensures that any differences in gene expression that arise from batch effects are minimized, making the data more comparable across studies.

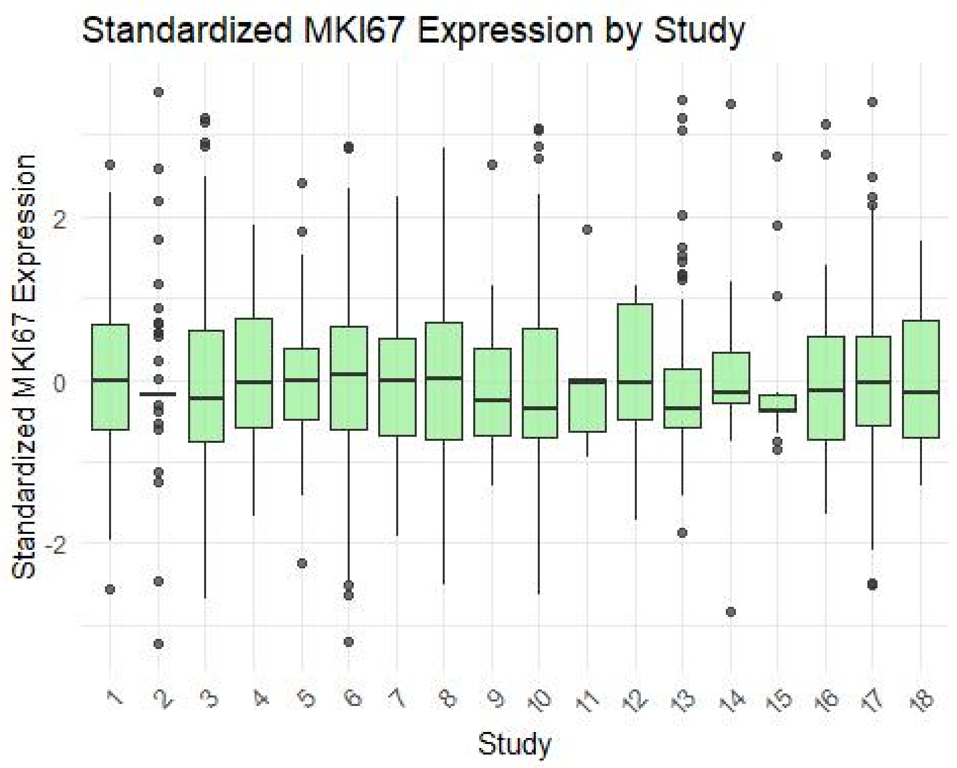
After the correction process, the data is split back into individual studies, preserving the original structure of each study’s data. It is essential to verify that this normalization process has not altered the dimensions of the data—meaning that the number of genes and samples remains consistent after normalization.

Finally, we visualize the results using boxplots to compare the gene expression distributions across the studies both before and after correcting for batch effects using MKI67. This side-by-side visualization allows us to assess the effectiveness of the correction process in making the studies more comparable. The goal is to ensure that any prior differences between the studies' distributions, which may have been caused by batch effects, are minimized, creating a more uniform dataset for further analysis.

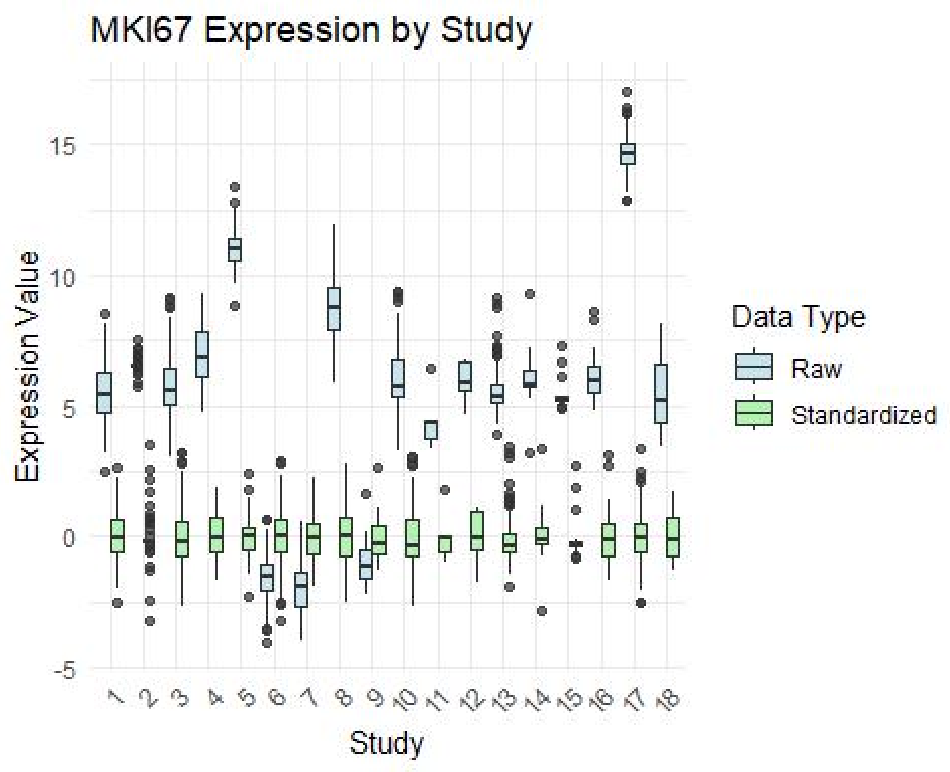
```r
# Function to standardize gene expression data
standardize_expression <- function(expression_values) {
  if(length(expression_values) > 1) {
    mean_val <- mean(expression_values, na.rm = TRUE)
    sd_val <- sd(expression_values, na.rm = TRUE)
    return((expression_values - mean_val) / sd_val)
  } else {
    return(NA)
  }
}

# Extract and standardize MKI67 expression across studies
extract_and_standardize_mki67 <- function(studies_list, gene_name = "MK
I67") {
  # Check if MKI67 exists in each study
  study_has_mki67 <- sapply(studies_list, function(study) gene_name %i
n% rownames(study))

  if(!all(study_has_mki67)) {
    warning(paste0("MKI67 not found in studies: ",
                   paste(which(!study_has_mki67), collapse = ", ")))
  }

  # Create data frames for both raw and standardized expression
  raw_data <- data.frame()
  standardized_data <- data.frame()

  for(i in seq_along(studies_list)) {
    study <- studies_list[[i]]

    if(gene_name %in% rownames(study)) {
      # Extract MKI67 expression values
      expression_values <- as.numeric(study[gene_name, ])

      # Standardize within this study
      standardized_values <- standardize_expression(expression_values)

      # Create and combine data frames
      study_raw_df <- data.frame(
        Study = i,
        Expression = expression_values,
        Type = "Raw"
      )
      study_std_df <- data.frame(
        Study = i,
        Expression = standardized_values,
        Type = "Standardized"
      )

      raw_data <- rbind(raw_data, study_raw_df)
      standardized_data <- rbind(standardized_data, study_std_df)
    }
  }

  # Combine both datasets for plotting
  combined_data <- rbind(raw_data, standardized_data)

  return(list(
    raw = raw_data,
    standardized = standardized_data,
    combined = combined_data
  ))
}

# Extract and standardize MKI67 across studies
mki67_data <- extract_and_standardize_mki67(XX_pCR)

# Create plots
plot_boxplots <- function(data) {
  # Plot raw expression
  raw_plot <- ggplot(data$raw, aes(x = factor(Study), y = Expression))
+
    geom_boxplot(fill = "lightblue", alpha = 0.7) +
    theme_minimal() +
    labs(title = "Raw MKI67 Expression by Study",
         x = "Study",
         y = "MKI67 Expression") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1))

  # Plot standardized expression
  std_plot <- ggplot(data$standardized, aes(x = factor(Study), y = Expr
ession)) +
    geom_boxplot(fill = "lightgreen", alpha = 0.7) +
    theme_minimal() +
    labs(title = "Standardized MKI67 Expression by Study",
         x = "Study",
         y = "Standardized MKI67 Expression") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1))

  # Plot both in one visualization if you have the gridExtra package
  # library(gridExtra)
  # grid.arrange(raw_plot, std_plot, ncol = 1)

  return(list(raw_plot = raw_plot, std_plot = std_plot))
}

# Generate plots
mki67_plots <- plot_boxplots(mki67_data)

# Display raw expression plot
print(mki67_plots$raw_plot)
```



```
# Display standardized expression plot
print(mki67_plots$std_plot)
```



```r
# Create a combined plot showing both raw and standardized
combined_plot <- ggplot(mki67_data$combined, aes(x = factor(Study), y =
 Expression, fill = Type)) +
  geom_boxplot(alpha = 0.7, position = position_dodge(width = 0.8)) +
  scale_fill_manual(values = c("Raw" = "lightblue", "Standardized" = "l
ightgreen")) +
  theme_minimal() +
  labs(title = "MKI67 Expression by Study",
       x = "Study",
       y = "Expression Value",
       fill = "Data Type") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

# Display combined plot
print(combined_plot)
```



SECTION 3: FEATURE SELECTION

To reduce the dimensionality of the more than 10,000 gene features while preserving discriminative signal, I designed a two-stage feature selection pipeline combining univariate gene filtering and principal component analysis (PCA).

3.1 Rationale for AUC Threshold Selection

To avoid overfitting and eliminate noisy features, I first filter genes by their ability to distinguish responders from non-responders using area under the ROC curve (AUC).

Each gene’s direction of association with response is recorded by comparing the mean expression in responders versus non-responders. If a gene has inverse discrimination (AUC < 0.5), we flip its sign and treat it as predictive in the opposite direction.

3.2 PCA Feature Construction

Once genes pass the AUC threshold, I use PCA to extract up to k uncorrelated components from the signed gene expression matrix. This step summarizes expression patterns while reducing noise and multicollinearity. Missing gene values are imputed by the gene’s mean across samples before PCA.

3.3 Cross-Validation Procedure

The function CV_pca performs leave-one-study-out cross-validation to jointly select the optimal gene filtering threshold in terms of AUC (α) and the number of PCA components (k). For each fold:

We compute AUCs and p-values on the training data to select genes. PCA is applied to the selected gene set. A logistic regression model is trained and validated on held-out study data. AUCs are averaged across folds to construct a performance heatmap.

The optimal pair (α, k) is chosen based on the highest mean validation AUC.

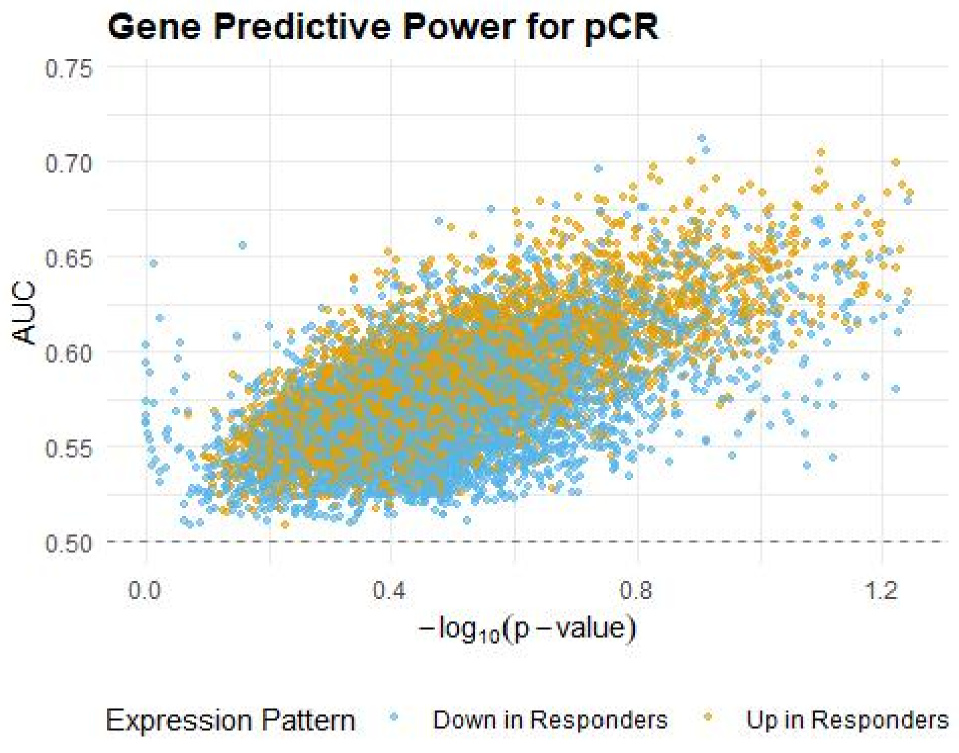
The CompScores function calculates these predictive metrics for each gene by: 1) Computing the t-test statistic comparing expression in responders vs non-responders 2) Converting this to a p-value 3) Calculating the negative log p-value for visualization purposes 4) Computing the AUC to quantify discriminative ability

For each gene that meets our threshold, we also record whether its AUC is above or below 0.5, indicating positive or negative association with response (this information becomes crucial in our later weighting scheme). This approach removes non-predictive genes while aligning with our evaluation metric of interest (AUC).

We generated a scatter plot showing the relationship between gene-level AUC and negative log p-value across all studies:

```
# AUC vs Negative Log P-value Analysis

# Fast AUC calculation function without requiring the pROC package
fast_auc = function(x, y) {
  # Handle missing or constant values
  if (length(unique(x[!is.na(x)])) <= 1 || all(is.na(x))) {
    return(NA)  # Return NA if there is no variance or if all values ar
e NA
  }

  # Handle missing values in y
  if (any(is.na(y))) {
    valid_idx <- !is.na(y)
    x <- x[valid_idx]
    y <- y[valid_idx]
  }

  # Ensure y is binary (0/1)
  if (!all(y %in% c(0, 1))) {
    return(NA)
  }

  # Handle case where all y values are the same
  if (length(unique(y)) == 1) {
    return(NA)
  }

  # Rank values of x
  rank_value = rank(x, ties.method="average", na.last="keep")
  n_pos = sum(y == 1, na.rm = TRUE)  # Number of responders
  n_neg = sum(y == 0, na.rm = TRUE)  # Number of non-responders

  # If either group is empty, return NA
  if (n_pos == 0 || n_neg == 0) {
    return(NA)
  }

  rank_sum = sum(rank_value[y == 1], na.rm = TRUE)
  u_value = rank_sum - (n_pos * (n_pos + 1)) / 2
  auc = u_value / (n_pos * n_neg)

  # Check if AUC is NA and return NA if true
  if (is.na(auc)) {
    return(NA)
  }

  # Return AUC or its complement (in case AUC is less than 0.5)
  if (auc >= 0.50) {
    return(auc)
  } else {
    return(1.0 - auc)
  }
}

# Helper function for fast t-test with robust error handling
fast_t_test = function(x, y) {
  # Handle missing values
  x <- x[!is.na(x)]
  y <- y[!is.na(y)]

  # Check if we have enough data to perform t-test
  if (length(x) < 3 || length(y) < 3) {
    return(list(p_value = 1, pooled_var = NA))
  }

  # Check for variance in both groups
  if (var(x) == 0 && var(y) == 0) {
    return(list(p_value = 1, pooled_var = 0))
```

```r
  }

  # Calculate statistics
  nx = length(x); mx = mean(x); vx = var(x)
  ny = length(y); my = mean(y); vy = var(y)

  # Avoid division by zero
  if (is.na(vx) || is.na(vy) || (vx == 0 && vy == 0)) {
    return(list(p_value = 1, pooled_var = NA))
  }

  # Calculate pooled standard error
  stderr <- sqrt(vx/nx + vy/ny)

  # If standard error is zero or NA, return p-value of 1
  if (is.na(stderr) || stderr == 0) {
    return(list(p_value = 1, pooled_var = NA))
  }

  # Calculate degrees of freedom using Welch-Satterthwaite equation
  df <- (vx/nx + vy/ny)^2 / ((vx/nx)^2/(nx-1) + (vy/ny)^2/(ny-1))

  # Handle edge cases
  if (is.na(df) || df <= 0) {
    df <- nx + ny - 2
  }

  # Calculate t-statistic
  tstat = (mx - my)/stderr

  # Calculate p-value
  p_value <- 2 * pt(-abs(tstat), df)

  # Return results
  return(list(p_value = p_value, pooled_var = stderr^2))
}

# CompScores function with robust error handling
CompScores <- function(XX_pCR, pCR) {
  # Verify data exists
  if (is.null(XX_pCR) || length(XX_pCR) == 0) {
    stop("XX_pCR data is empty or NULL")
  }

  if (is.null(pCR) || length(pCR) == 0) {
    stop("pCR data is empty or NULL")
  }

  # Print summary of data availability
  cat("Analyzing data across", length(XX_pCR), "studies\n")
  for (i in 1:length(XX_pCR)) {
    if (!is.null(XX_pCR[[i]]) && !is.null(pCR[[i]])) {
      cat("Study", i, "has", ncol(XX_pCR[[i]]), "samples and",
          sum(pCR[[i]] == 1, na.rm = TRUE), "responders\n")
    } else {
      cat("Study", i, "has missing data\n")
    }
  }

  # Get genes from first valid study
  first_valid_idx <- which(!sapply(XX_pCR, is.null))[1]
  if (is.na(first_valid_idx)) {
    stop("No valid study found")
  }

  gene_names <- rownames(XX_pCR[[first_valid_idx]])
  NGenes <- length(gene_names)
  cat("Found", NGenes, "genes\n")

  # Initialize result arrays
  AUC_matrix <- matrix(NA, nrow = NGenes, ncol = length(XX_pCR))
  nlpvalueT_matrix <- matrix(NA, nrow = NGenes, ncol = length(XX_pCR))
  FoldChange_matrix <- matrix(NA, nrow = NGenes, ncol = length(XX_pCR))

  # Set seed for reproducibility
  set.seed(5118)

  # Calculate metrics for each gene in each study
  for (study_idx in 1:length(XX_pCR)) {
    # Skip if study data is null
    if (is.null(XX_pCR[[study_idx]]) || is.null(pCR[[study_idx]])) {
      cat("Skipping study", study_idx, "due to missing data\n")
      next
    }

    study_data <- XX_pCR[[study_idx]]
    pcr_response <- pCR[[study_idx]]

    # Check validity of pCR data
    if (any(is.na(pcr_response))) {
      cat("Study", study_idx, "has", sum(is.na(pcr_response)), "NA valu
es in response data\n")
      # Remove NA values
      valid_idx <- !is.na(pcr_response)
      pcr_response <- pcr_response[valid_idx]
      study_data <- study_data[, valid_idx, drop = FALSE]
    }
    # Check if enough samples in each response group
    n_responders <- sum(pcr_response == 1, na.rm = TRUE)
    n_non_responders <- sum(pcr_response == 0, na.rm = TRUE)

    if (n_responders < 3 || n_non_responders < 3) {
      cat("Study", study_idx, "has insufficient samples in response gro
ups:",
          n_responders, "responders,", n_non_responders, "non-responder
s\n")
      next
    }

    # Calculate metrics for each gene
    for (gene_idx in 1:NGenes) {
      gene_expression <- study_data[gene_idx, ]

      # Skip genes with too many NAs
      na_count <- sum(is.na(gene_expression))
      if (na_count > 0.2 * length(gene_expression)) {
        next
      }

      # Calculate AUC
      AUC_matrix[gene_idx, study_idx] <- tryCatch({
        fast_auc(gene_expression, pcr_response)
      }, error = function(e) {
        cat("Error calculating AUC for gene", gene_idx, "in study", stu
dy_idx, ":", e$message, "\n")
        return(NA)
      })

      # Get expression values for responders and non-responders
      expr_responders <- gene_expression[pcr_response == 1]
      expr_non_responders <- gene_expression[pcr_response == 0]

      # Perform t-test
      t_test_result <- tryCatch({
        fast_t_test(expr_responders, expr_non_responders)
      }, error = function(e) {
        cat("Error in t-test for gene", gene_idx, "in study", study_idx,
 ":", e$message, "\n")
        return(list(p_value = 1, pooled_var = NA))
      })

      # Convert p-value to negative log p-value (with minimum p-value c
ap to avoid Inf)
      min_p <- 1e-10  # Minimum p-value to avoid -log(p) becoming too l
arge
      p_val <- max(t_test_result$p_value, min_p)
      nlpvalueT_matrix[gene_idx, study_idx] <- -log10(p_val)  # Use log
10 for better scaling

      # Calculate fold change
      FoldChange_matrix[gene_idx, study_idx] <- mean(expr_responders, n
a.rm = TRUE) -
                                                mean(expr_non_responder
s, na.rm = TRUE)
    }
  }

  # Aggregate results across studies (using median to be robust to outl
iers)
  AUC_aggregated <- apply(AUC_matrix, 1, median, na.rm = TRUE)
  nlpvalueT_aggregated <- apply(nlpvalueT_matrix, 1, median, na.rm = TR
UE)
  FoldChange_aggregated <- apply(FoldChange_matrix, 1, median, na.rm =
TRUE)

  # Calculate variance of AUC across studies (measure of consistency)
  AUC_variance <- apply(AUC_matrix, 1, var, na.rm = TRUE)

  # Combine results into a data frame
  scores <- data.frame(
    GeneID = gene_names,
    AUC = AUC_aggregated,
    nlpvalueT = nlpvalueT_aggregated,
    FoldChange = FoldChange_aggregated,
    AUC_variance = AUC_variance
  )

  # Remove rows with NA values
  valid_rows <- !is.na(scores$AUC) & !is.na(scores$nlpvalueT)
  cat("Keeping", sum(valid_rows), "out of", nrow(scores), "genes with v
alid data\n")
  scores <- scores[valid_rows, ]

  return(scores)
}

# Function to create plot with error handling
create_plot <- function(scores_df) {
  # Check if we have data to plot
  if (nrow(scores_df) == 0) {
    stop("No valid data to plot")
  }

  # Print summary statistics
  cat("\nSummary of AUC values:\n")
  print(summary(scores_df$AUC))
  cat("\nSummary of -log10(p-values):\n")
  print(summary(scores_df$nlpvalueT))

  # Create direction column based on fold change
  scores_df$direction <- ifelse(scores_df$FoldChange > 0, "Up in Respon
ders", "Down in Responders")

  # Determine reasonable axis limits
  max_nlp <- min(max(scores_df$nlpvalueT, na.rm = TRUE),
                 quantile(scores_df$nlpvalueT, 0.99, na.rm = TRUE) * 1.
1)
  max_auc <- min(max(scores_df$AUC, na.rm = TRUE), 0.95)  # Cap at 0.95
 for better visualization

  # Create a simple yet informative scatter plot
  p <- ggplot(scores_df, aes(x = nlpvalueT, y = AUC, color = direction))
 +
    geom_point(alpha = 0.6, size = 1.0) +
    scale_color_manual(values = c("Up in Responders" = "#E69F00",
                                  "Down in Responders" = "#56B4E9")) +
    labs(
      title = "Gene Predictive Power for pCR",
      x = expression(-log[10](p-value)),
      y = "AUC",
      color = "Expression Pattern"
    ) +
    theme_minimal() +
    theme(
      plot.title = element_text(face = "bold"),
      legend.position = "bottom"
    ) +
    # Add reference lines
    geom_hline(yintercept = 0.5, linetype = "dashed", color = "gray50")
 +
    geom_vline(xintercept = -log10(0.05), linetype = "dashed", color =
"gray50") +
    # Set reasonable axis limits
    scale_x_continuous(limits = c(0, max_nlp)) +
    scale_y_continuous(limits = c(0.5, max_auc))

  return(p)
}

# Try to run with error handling
tryCatch({
  # Run the robust score computation
  robust_scores <- CompScores(XX_pCR, pCR)
  # Create and display the plot
  robust_plot <- create_plot(robust_scores)
  print(robust_plot)

  # Optional: Save the plot to a file
  ggsave("auc_vs_pvalue_plot.png", robust_plot, width = 8, height = 6,
dpi = 300)

  # Return the scores dataframe for further analysis
  cat("Analysis complete. Scores data frame is available for further us
e.\n")

}, error = function(e) {
  cat("Error in analysis:", e$message, "\n")
})

## Analyzing data across 18 studies
## Study 1 has 114 samples and 16 responders
## Study 2 has 53 samples and 37 responders
## Study 3 has 261 samples and 55 responders
## Study 4 has 28 samples and 5 responders
## Study 5 has 31 samples and 5 responders
## Study 6 has 128 samples and 32 responders
## Study 7 has 20 samples and 3 responders
## Study 8 has 122 samples and 20 responders
## Study 9 has 16 samples and 0 responders
## Study 10 has 221 samples and 43 responders
## Study 11 has 6 samples and 0 responders
## Study 12 has 17 samples and 5 responders
## Study 13 has 71 samples and 12 responders
## Study 14 has 25 samples and 4 responders
## Study 15 has 16 samples and 2 responders
## Study 16 has 54 samples and 19 responders
## Study 17 has 115 samples and 26 responders
## Study 18 has 11 samples and 2 responders
## Found 10115 genes
## Study 6 has 5 NA values in response data
## Study 9 has insufficient samples in response groups: 0 responders, 1
6 non-responders
## Study 11 has insufficient samples in response groups: 0 responders,
6 non-responders
## Study 14 has 1 NA values in response data
## Study 15 has insufficient samples in response groups: 2 responders,
14 non-responders
## Study 18 has insufficient samples in response groups: 2 responders,
9 non-responders
## Keeping 10115 out of 10115 genes with valid data
##
## Summary of AUC values:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.5085  0.5538  0.5733  0.5769  0.5957  0.7413
##
## Summary of -log10(p-values):
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.3543  0.4631  0.4914  0.5918  2.2185

## Warning: Removed 48 rows containing missing values or values outside
 the scale range
## (`geom_point()`).

## Warning: Removed 1 row containing missing values or values outside t
he scale range
## (`geom_vline()`).

## Warning: Removed 48 rows containing missing values or values outside
 the scale range
## (`geom_point()`).

## Warning: Removed 1 row containing missing values or values outside t
he scale range
## (`geom_vline()`).
```



```
## Analysis complete. Scores data frame is available for further use.
```

As visualized in Figure: CV Validation AUC Heatmap, we conducted a comprehensive grid search across AUC thresholds (α) ranging from 0.525 to 0.650 in increments of 0.025. This range was selected, based on the scatterplot **Gene Predictive Power for pCR** generated above, to balance sensitivity and specificity: lower thresholds risk introducing noise by admitting weak or non-informative genes, while higher thresholds may eliminate potentially useful but moderately predictive genes. By tuning α jointly with the number of principal components (k) in a leave-one-study-out (LOSO) cross-validation framework, we aimed to find the sweet spot where model generalization is optimized. The resulting heatmap visualizes validation AUCs across different combinations, guiding the selection of (α, k) that maximized cross-study performance.

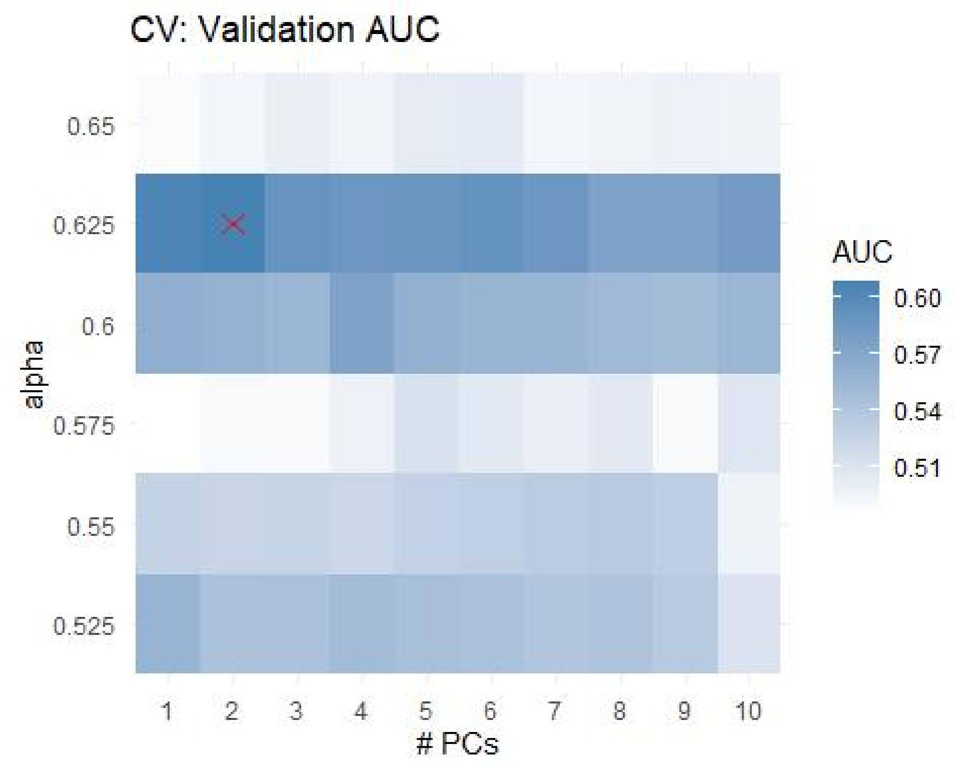
Feature Selection

To transform the selected gene set into a compact and informative feature representation, I applied Principal Component Analysis (PCA). After filtering genes using AUC and p-value thresholds and recording their direction of association with pCR, I applied sign correction (flipping negatively associated genes) to align expression directions.

Then, PCA was performed on the signed expression matrix to extract the top k uncorrelated components. These components summarize the dominant axes of variation across the selected genes, reduce noise, and help mitigate multicollinearity. Unlike individual gene expression values, PCA-derived features are orthogonal and tend to generalize better across heterogeneous studies, especially when followed by classifiers like logistic regression or GBM. I did this to ensure that downstream models are trained on signal-rich, low-dimensional inputs that remain interpretable and robust to overfitting.

```r
get_predictors_pca <- function(XX_sel, signs, k) {
  if (nrow(XX_sel) == 0)
    return(matrix(nrow = 0, ncol = ncol(XX_sel)))

  Pd <- t(XX_sel * signs)
  Pd <- apply(Pd, 2, function(v) { v[is.na(v)] <- mean(v, na.rm = TRUE);
 v })

  var_cols <- apply(Pd, 2, var, na.rm = TRUE)
  non_const_cols <- which(var_cols > 1e-10)

  if (length(non_const_cols) == 0) {
    warning("All columns have zero variance!")
    return(matrix(nrow = 0, ncol = ncol(Pd)))
  }
  Pd <- Pd[, non_const_cols, drop = FALSE]
  k_actual <- min(k, ncol(Pd), nrow(Pd) - 1)
  if (k_actual < 1)
    return(matrix(nrow = 0, ncol = ncol(Pd)))

  pca <- prcomp(Pd, center = TRUE, scale. = TRUE)
  comps <- pca$x[, seq_len(k_actual), drop = FALSE]

  t(comps)
}

# ===============================
# 2. Fast AUC function
# ===============================
fast_auc <- function(scores, labels) {
  ok <- !is.na(scores) & !is.na(labels)
  scores <- scores[ok]; labels <- labels[ok]
  if (length(unique(scores)) < 2 || length(unique(labels)) < 2) return
(NA_real_)
  r   <- rank(scores, ties.method = "average")
  n1  <- sum(labels == 1); n0 <- sum(labels == 0)
  if (n1 == 0 || n0 == 0) return(NA_real_)
  u   <- sum(r[labels == 1]) - n1 * (n1 + 1) / 2
  auc <- u / (n1 * n0)
  if (auc < 0.5) 1 - auc else auc
}

# ===============================
# 3. Gene selection by AUC and t-test
# ===============================
threshold_with_sign <- function(XX, YY, alpha, p_thr = 0.05) {
  G <- nrow(XX)
  aucs <- numeric(G); pvals <- numeric(G); dirs <- integer(G)

  for (i in seq_len(G)) {
    g <- XX[i, ]
    aucs[i] <- fast_auc(g, YY)
    m1 <- mean(g[YY == 1], na.rm = TRUE)
    m0 <- mean(g[YY == 0], na.rm = TRUE)
    dir <- if (m1 >= m0) 1L else -1L
    if (!is.na(aucs[i]) && aucs[i] < 0.5) {
      aucs[i] <- 1 - aucs[i]
      dir <- -dir
    }
    dirs[i] <- dir

    g1 <- na.omit(g[YY == 1])
    g0 <- na.omit(g[YY == 0])
    pvals[i] <- if (length(g1) > 1 && length(g0) > 1 && var(g1) > 0 &&
var(g0) > 0) {
      t.test(g1, g0)$p.value
    } else {
      1
    }
  }

  sel <- which(!is.na(aucs) & aucs >= alpha & pvals <= p_thr)

  if (length(sel) == 0) {
    return(list(
      genes = matrix(nrow = 0, ncol = ncol(XX)),
      signs = integer(0),
      auc_values = numeric(0),
      p_values = numeric(0)
    ))
  }

  list(
    genes = XX[sel, , drop = FALSE],
    signs = dirs[sel],
    auc_values = aucs[sel],
    p_values = pvals[sel]
  )
}

# ================================
# 4. Cross-validation to pick best alpha and k
# ================================
CV_pca <- function(XX_pCR, pCR, alphas, kmax) {
  library(foreach)
  library(doParallel)
  library(ROCR)
  library(ggplot2)

  cores <- min(parallel::detectCores() - 1, 4)
  cl <- makeCluster(cores)
  registerDoParallel(cl)
  on.exit(stopCluster(cl), add = TRUE)

  XX <- NULL
  YYdf <- data.frame()
  genes <- NULL

  for (i in seq_along(XX_pCR)) {
    expr_mat <- XX_pCR[[i]]; resp <- pCR[[i]]
    if (is.null(expr_mat) || all(is.na(resp))) next
    ok <- !is.na(resp)
    expr <- t(scale(t(expr_mat[, ok, drop = FALSE])))
    if (is.null(XX)) {
      genes <- rownames(expr)
      XX <- expr
    } else {
      genes <- intersect(genes, rownames(expr))
      XX <- cbind(XX[genes, , drop = FALSE], expr[genes, , drop = FALS
E])
    }
    YYdf <- rbind(YYdf, data.frame(response = resp[ok], study = i))
  }

  if (is.null(XX)) stop("No valid data.")
  XX[is.na(XX)] <- 0
  folds <- unique(YYdf$study)

  fold_list <- foreach(f = folds, .combine = list, .multicombine = TRUE,
                       .packages = c("ROCR"),
                       .export = c("fast_auc", "threshold_with_sign", "
get_predictors_pca")) %dopar% {
    train_idx <- YYdf$study != f
    test_idx  <- !train_idx
    Xtr <- XX[, train_idx, drop = FALSE]
    Ytr <- YYdf$response[train_idx]
    Xv  <- XX[, test_idx, drop = FALSE]
    Yv  <- YYdf$response[test_idx]

    # Skip if not binary labels
    if (length(unique(Ytr)) != 2 || length(unique(Yv)) != 2) {
      cat(sprintf("Skipping fold %d: Not enough classes\n", f))
      return(matrix(NA_real_, length(alphas), kmax))
    }

    M <- matrix(NA_real_, length(alphas), kmax)
    for (ai in seq_along(alphas)) {
      sel <- threshold_with_sign(Xtr, Ytr, alphas[ai])
      if (nrow(sel$genes) == 0) next

      keep_genes <- intersect(rownames(sel$genes), rownames(Xv))
      if (length(keep_genes) == 0) next

      Gtr <- sel$genes[keep_genes, , drop = FALSE]
      Gv  <- Xv[keep_genes, , drop = FALSE]
      S   <- sel$signs[match(keep_genes, rownames(sel$genes))]

      Ptr <- get_predictors_pca(Gtr, S, kmax)
      Pv  <- get_predictors_pca(Gv,  S, kmax)

      for (k in seq_len(min(nrow(Ptr), kmax))) {
        df_tr <- data.frame(t(Ptr[1:k, , drop = FALSE]), response = fac
```

```r
tor(Ytr))
        mdl   <- tryCatch(glm(response ~ ., data = df_tr, family = bino
mial()), error = function(e) NULL)
        if (is.null(mdl)) next

        preds <- tryCatch(predict(mdl, newdata = data.frame(t(Pv[1:k, ,
 drop = FALSE])), type = "response"), error = function(e) NULL)
        if (is.null(preds) || any(is.na(preds))) next

        perf <- tryCatch(ROCR::performance(ROCR::prediction(preds, Yv),
 "auc"), error = function(e) NULL)
        if (!is.null(perf)) M[ai, k] <- perf@y.values[[1]]
      }
    }
    M
  }

  arr <- simplify2array(fold_list)
  avg <- apply(arr, c(1,2), mean, na.rm = TRUE)

  grid <- expand.grid(alpha = alphas, k = seq_len(kmax))
  grid$auc <- as.vector(avg)

  best_idx <- which.max(grid$auc)
  alpha_best <- grid$alpha[best_idx]
  k_best     <- grid$k[best_idx]

  heatmap <- ggplot(grid, aes(x = factor(k), y = factor(alpha), fill =
auc)) +
    geom_tile() +
    geom_point(data = grid[best_idx, , drop = FALSE], aes(x = factor(k),
 y = factor(alpha)),
               color = "red", size = 3, shape = 4) +
    scale_fill_gradient(low = "white", high = "steelblue", na.value = "
grey90") +
    labs(title = "CV: Validation AUC", x = "# PCs", y = "alpha", fill =
 "AUC") +
    theme_minimal()

  list(
    alpha_best = alpha_best,
    k_best = k_best,
    heatmap = heatmap,
    avg_results = avg
  )
}

# ===============================
# Example call
# ================================
cv_result <- CV_pca(XX_pCR, pCR, alphas = seq(0.525, 0.650, 0.025), kma
x = 10)
print(cv_result$heatmap)
```



As shown in the Heatmap above, the highest validation AUC is achieved when using alpha = 0.625 as the gene filtering threshold and 2 principal components (PCs). This grid search balances model complexity and signal retention.

We also see that the dataset spans 18 studies, with a wide range in size and class balance. For instance, Study 3 includes 261 samples and 55 responders, while smaller studies such as Study 11 (6 samples, 0 responders) or Study 9 (16 samples, 0 responders) provide insufficient signal and were excluded from modeling. Across all studies, we initially considered 10,115 genes, all of which passed the quality filter. However, only genes meeting AUC and p-value thresholds contributed to predictor construction.

SECTION 4: MODELS, RESULTS

Now that we have obtained the optimal α and $\k$ values using cross-validation, it is time to dive into the motivation behind my classifier choice.

We began with logistic regression as a baseline due to its simplicity, interpretability, and widespread use in clinical settings. It provides a clear benchmark and enables direct comparison against the PAM50-based logistic regression. Instead of using LASSO for feature selection, we opted to pre-filter genes based on AUC and p-value, followed by PCA-based compression, which reduced the dimensionality while preserving signal and avoiding the instability often associated with penalized regression on high-dimensional data. Random forest was chosen next to capture non-linear relationships and interactions between the selected features, while maintaining robustness to overfitting. Finally, we included gradient boosting (GBM) for its strong performance in many biomedical prediction tasks and its ability to optimize over complex feature interactions with regularization via shrinkage and tree depth. This progression allowed us to assess how model complexity impacts performance while keeping the feature engineering process consistent.

```r
# Libraries
library(randomForest)
library(gbm)
library(ROCR)
library(knitr)

# Function to get predictor with signs
get_predictors_with_sign <- function(XX_cut, groups, k, signs) {
  # Apply directionality from AUC
  XX_signed <- XX_cut * signs

  # Initialize predictors matrix correctly with samples as rows
  n_samples <- ncol(XX_cut)
  predictors <- matrix(0, nrow = n_samples, ncol = k)
  rownames(predictors) <- colnames(XX_cut)
  colnames(predictors) <- paste0("Feat", 1:k)

  if (length(groups) >= k) {
    # Average each top group's signed expression
    for (i in 1:k) {
      group_genes <- groups[[i]]
      if (length(group_genes) > 0) {
        predictors[, i] <- colMeans(XX_signed[group_genes, , drop = FAL
SE])
      }
    }
  } else {
    # Fallback to PCA without transposing the result
    signed_mat <- XX_cut * signs
    Pd <- t(signed_mat)  # Samples × genes
    Pd <- apply(Pd, 2, function(v) {
      v[is.na(v)] <- mean(v, na.rm = TRUE)
      v
    })
    k_actual <- min(k, ncol(Pd), nrow(Pd) - 1)
    pca <- tryCatch(
      prcomp(Pd, center = TRUE, scale. = TRUE),
      error = function(e) NULL
    )
    if (!is.null(pca) && k_actual >= 1) {
      predictors[, 1:k_actual] <- pca$x[, 1:k_actual, drop = FALSE]
    }
  }

  return(predictors)
}

# Function to calculate AUC safely (handling NA values)
auc_safe <- function(preds, truth) {
  ok <- !is.na(preds) & !is.na(truth)
  if (sum(ok) < 1) return(NA)
  performance(prediction(preds[ok], truth[ok]), "auc")@y.values[[1]]
}

# Function to calculate AUC quickly
fast_auc <- function(x, y) {
  x1 <- x[y == 1]
  x2 <- x[y == 0]

  # Count pairs where x1 > x2, x1 = x2, x1 < x2
  n1 <- length(x1)
  n2 <- length(x2)

  if (n1 == 0 || n2 == 0) return(0.5)

  r <- rank(c(x1, x2))
  auc <- (sum(r[1:n1]) - n1 * (n1 + 1) / 2) / (n1 * n2)
  return(auc)
}

# Function to select genes meeting the AUC threshold and determine thei
r direction
threshold_with_sign <- function(XX_train, YY_train, alpha_threshold, p_
threshold = 0.05) {
  NGenes <- nrow(XX_train)
  AUC_values <- numeric(NGenes)
  p_values <- numeric(NGenes)
  directions <- numeric(NGenes)
  # Calculate AUC and sign for each gene
  for (i in 1:NGenes) {
    gene_expr <- XX_train[i, ]

    # Skip genes with no variance
    if (length(unique(gene_expr)) <= 1) {
      AUC_values[i] <- 0.5
      p_values[i] <- 1
      directions[i] <- 0
      next
    }

    # Calculate AUC
    pred <- prediction(gene_expr, YY_train)
    auc_result <- performance(pred, "auc")@y.values[[1]]

    # Determine if gene is positively or negatively associated with res
ponse
    mean_resp <- mean(gene_expr[YY_train == 1])
    mean_non_resp <- mean(gene_expr[YY_train == 0])
    direction <- ifelse(mean_resp > mean_non_resp, 1, -1)

    # Store the direction that makes AUC > 0.5
    if (auc_result < 0.5) {
      auc_result <- 1 - auc_result
      direction <- -direction
    }

    AUC_values[i] <- auc_result
    directions[i] <- direction

    # Calculate p-value using t-test
    t_test <- t.test(gene_expr[YY_train == 1], gene_expr[YY_train == 0])
    p_values[i] <- t_test$p.value
  }

  # Select genes that meet the AUC and p-value thresholds
  selected_genes <- which(AUC_values >= alpha_threshold & p_values <= p
_threshold)

  # Return selected genes and their directions
  if (length(selected_genes) > 0) {
    return(list(
      genes = XX_train[selected_genes, , drop = FALSE],
      signs = directions[selected_genes],
      auc_values = AUC_values[selected_genes],
      p_values = p_values[selected_genes]
    ))
  } else {
```

```r
    # Return empty matrices/vectors if no genes meet the criteria
    return(list(
      genes = matrix(nrow = 0, ncol = ncol(XX_train)),
      signs = numeric(0),
      auc_values = numeric(0),
      p_values = numeric(0)
    ))
  }
}

# Function to cluster genes based on correlation
corr_cluster <- function(expr_mat, corr_thr = 0.8) {
  # Calculate the pairwise correlation matrix
  cor_mat <- cor(t(expr_mat), method = "pearson", use = "pairwise.compl
ete.obs")

  # Take absolute values of correlations for clustering purposes
  abs_cor_mat <- abs(cor_mat)

  # Create a distance matrix where distance = 1 - abs(correlation)
  dist_mat <- as.dist(1 - abs_cor_mat)

  # Perform hierarchical clustering using average linkage
  hc <- hclust(dist_mat, method = "average")

  # Cut the dendrogram at a height corresponding to the threshold
  cluster_assignment <- cutree(hc, h = 1 - corr_thr)

  # Organize genes into groups
  groups <- split(names(cluster_assignment), cluster_assignment)

  # Sort groups by size (largest first)
  group_sizes <- sapply(groups, length)
  groups_sorted <- groups[order(group_sizes, decreasing = TRUE)]

  return(groups_sorted)
}


# Function to fit models (LR, RF, GBM)
fit_model <- function(predictors, YY_train, indexes, k, lr = FALSE, rf
= FALSE, gbm = FALSE) {
  if (lr) {
    df_train <- as.data.frame(predictors[indexes, ])
    names(df_train) <- paste0("P", 1:k)
    model <- glm(YY_train ~ ., data = df_train, family = binomial(link
= "logit"))
    return(model)
  } else if (rf) {
    model <- randomForest(
      x = as.matrix(predictors[indexes, ]),
      y = as.factor(YY_train),
      ntree = 500,
      mtry = floor(sqrt(k))
    )
    return(model)
  } else if (gbm) {
    # Convert target to 0/1 if it's not already
    YY_train_01 <- ifelse(YY_train == TRUE | YY_train == 1, 1, 0)

    # Create training data frame
    df_train <- as.data.frame(predictors[indexes, ])
    names(df_train) <- paste0("P", 1:k)
    df_train$target <- YY_train_01

    # Train GBM model with sensible defaults
    model <- gbm(
      formula = target ~ .,
      data = df_train,
      distribution = "bernoulli", # For binary classification
      n.trees = 500, # Number of trees
      interaction.depth = 3, # Depth of trees
      shrinkage = 0.01, # Learning rate
      cv.folds = 5, # Cross-validation for optimal tree number
      n.minobsinnode = 10, # Minimum observations in terminal nodes
      verbose = FALSE
    )

    # Find optimal number of trees using cross-validation
    best_trees <- gbm.perf(model, method = "cv", plot.it = FALSE)

    # Update model with best number of trees
    model$best_trees <- best_trees
    return(model)
  }
}

# Function to benchmark PAM50 predictor
# Improved benchmark function to handle PAM50/Prosigna predictions
benchmark <- function(pam50_preds, true_response, j) {
  # Create vectors to hold all predictions and responses
  all_preds <- vector("numeric")
  all_resp <- vector("numeric")
  study_idx <- vector("numeric")

  # Collect all predictions and responses with study indices
  for (i in seq_along(true_response)) {
    if (!is.null(pam50_preds[[i]]) && !is.null(true_response[[i]])) {
      valid_indices <- !is.na(pam50_preds[[i]]) & !is.na(true_response
[[i]])
      if (sum(valid_indices) > 0) {
        all_preds <- c(all_preds, pam50_preds[[i]][valid_indices])
        all_resp <- c(all_resp, true_response[[i]][valid_indices])
        study_idx <- c(study_idx, rep(i, sum(valid_indices)))
      }
    }
  }

  # Check if we have enough data
  if (length(all_preds) == 0 || length(unique(all_resp)) < 2) {
    return(c(NA, NA))
  }

  # Calculate train and test AUCs
  train_indices <- study_idx != j
  test_indices <- study_idx == j

  # Make sure both training and test sets have data from both classes
  train_valid <- length(unique(all_resp[train_indices])) == 2 && sum(tr
ain_indices) >= 2
  test_valid <- length(unique(all_resp[test_indices])) == 2 && sum(test
_indices) >= 2

  train_auc <- if(train_valid) auc_safe(all_preds[train_indices], all_r
esp[train_indices]) else NA
  test_auc <- if(test_valid) auc_safe(all_preds[test_indices], all_resp
[test_indices]) else NA

  return(c(train_auc, test_auc))
}
# benchmark <- function(pam50_preds, true_response, j) {
#   # Extract predictions for all studies
#   all_preds <- unlist(pam50_preds)
#   all_resp <- unlist(true_response)
#
#   # Identify which samples are from study j (test set)
#   study_idx <- rep(seq_along(true_response), sapply(true_response, le
ngth))
#
#   # Calculate train and test AUCs
#   train_auc <- auc_safe(all_preds[study_idx != j], all_resp[study_idx
 != j])
#   test_auc <- auc_safe(all_preds[study_idx == j], all_resp[study_idx
== j])
#
#   return(c(train_auc, test_auc))
# }
```

```r
# 1. Find common PAM50 genes across studies 4 to 16
pam50_gene_names <- Reduce(intersect, lapply(XX_pCR[4:16], function(exp
r) {
  if (is.null(expr)) character(0) else rownames(expr)
}))

# 2. Generate PAM50 predictions (normalized average expression)
pam50_pCR <- lapply(XX_pCR, function(expr_mat) {
  if (is.null(expr_mat)) return(rep(NA, ncol(expr_mat)))
  genes_present <- intersect(pam50_gene_names, rownames(expr_mat))
  if (length(genes_present) == 0) return(rep(NA, ncol(expr_mat)))

  expr_subset <- expr_mat[genes_present, , drop = FALSE]
  scores <- colMeans(expr_subset, na.rm = TRUE)

  # Normalize to [0, 1] range for interpretability
  scores <- (scores - min(scores, na.rm = TRUE)) /
            (max(scores, na.rm = TRUE) - min(scores, na.rm = TRUE))
  return(scores)
})



# Main workflow function v3
run_model_comparison <- function(XX_pCR, pCR, pam50_pCR, j, alpha_best
= 0.625, k_best = 2) {
  # 1) Combine & standardize across studies
  YY <- data.frame()
  XX <- NULL
  for(i in seq_along(XX_pCR)){
    mat <- XX_pCR[[i]]
    resp <- pCR[[i]]
    if(is.null(mat)) next
    ok <- !is.na(resp)
    if(!any(ok)) next
    mat_s <- t(scale(t(mat[,ok,drop=FALSE])))
    if(is.null(XX)) XX <- mat_s
    else {
      common <- intersect(rownames(XX), rownames(mat_s))
      XX <- cbind(XX[common,], mat_s[common,])
    }
    YY <- rbind(YY, data.frame(response=resp[ok], study=i))
  }
  XX[is.na(XX)] <- 0

  # 2) Train/test split
  idx_tr <- which(YY$study != j)
  idx_te <- which(YY$study == j)

  # Check if test set has both classes
  if (length(unique(YY$response[idx_te])) != 2) {
    warning(sprintf("Study %d: test set has only one class — skipping e
valuation", j))
    results <- matrix(NA, nrow=4, ncol=2)
    rownames(results) <- c("PAM50 benchmark","Logistic Regression","Ran
dom Forest","GBM")
    colnames(results) <- c("Train AUC","Test AUC")
    return(results)
  }

  # 3) Feature selection on training only
  sel <- threshold_with_sign(XX[,idx_tr,drop=FALSE], YY$response[idx_t
r], alpha_best)
  if(nrow(sel$genes)==0){
    warning("no genes passed; taking top 100 by AUC")
    aucs <- apply(XX[,idx_tr],1,fast_auc,YY$response[idx_tr])
    top  <- order(aucs,decreasing=TRUE)[1:100]
    sel  <- list(
      genes = XX[top,idx_tr,drop=FALSE],
      signs = setNames(ifelse(aucs[top]>=0.5,1,-1), rownames(XX)[top])
    )
  }
  names(sel$signs) <- rownames(sel$genes)

  # 4) Cluster & build full-sample predictor matrix
  groups      <- corr_cluster(sel$genes, corr_thr=1.0)
  genes_sel   <- rownames(sel$genes)
  signs_sel   <- sel$signs[genes_sel]
  XX_cut_full <- XX[genes_sel,,drop=FALSE]
  P           <- get_predictors_with_sign(XX_cut_full, groups, k_best,
signs_sel)

  stopifnot(nrow(P)==nrow(YY))

  # 5) Train models and calculate AUCs
  # Logistic Regression
  df_tr <- as.data.frame(P[idx_tr,])
  df_te <- as.data.frame(P[idx_te,])
  colnames(df_tr) <- colnames(df_te) <- paste0("P", 1:k_best)

  # Logistic Regression
  lr_mod <- fit_model(P, YY$response[idx_tr], idx_tr, k_best, lr=TRUE)
  lr_train_auc <- if(!is.null(lr_mod))
    auc_safe(
      predict(lr_mod, newdata=df_tr, type="response"),
      YY$response[idx_tr]
    ) else NA
  lr_test_auc <- if(!is.null(lr_mod))
    auc_safe(
      predict(lr_mod, newdata=df_te, type="response"),
      YY$response[idx_te]
    ) else NA

  # Random Forest
  rf_mod <- randomForest(
    x = df_tr,
    y = as.factor(YY$response[idx_tr]),
    ntree = 500,
    mtry = floor(sqrt(k_best)),
    maxnodes = 10,
    nodesize = 5
  )

  rf_train_probs <- predict(rf_mod, newdata = df_tr, type = "prob")[,2]
  rf_test_probs <- predict(rf_mod, newdata = df_te, type = "prob")[,2]
  rf_train_auc <- auc_safe(rf_train_probs, YY$response[idx_tr])
  rf_test_auc <- auc_safe(rf_test_probs, YY$response[idx_te])

  # GBM
  gbm_mod <- gbm::gbm(
    formula = response ~ .,
    data = data.frame(response = YY$response[idx_tr], df_tr),
    distribution = "bernoulli",
    n.trees = 100,
    interaction.depth = 3,
    shrinkage = 0.05,
    n.minobsinnode = 5,
    verbose = FALSE
  )

  best_iter <- gbm::gbm.perf(gbm_mod, method = "OOB", plot.it = FALSE)
  gbm_train_probs <- predict(gbm_mod, newdata = df_tr, n.trees = best_i
ter, type = "response")
  gbm_test_probs <- predict(gbm_mod, newdata = df_te, n.trees = best_it
er, type = "response")
  gbm_train_auc <- auc_safe(gbm_train_probs, YY$response[idx_tr])
  gbm_test_auc <- auc_safe(gbm_test_probs, YY$response[idx_te])

  # PAM50 benchmark - use improved benchmark function
  pam_res <- benchmark(pam50_pCR, pCR, j)
  # Add diagnostic information for debugging PAM50
  if(is.na(pam_res[1])) {
    cat("DEBUG: PAM50 training AUC is NA\n")
    # Check if we have PAM50 predictions for study j
    if(j <= length(pam50_pCR) && !is.null(pam50_pCR[[j]])) {
      cat(sprintf("Study %d has %d PAM50 predictions, %d are NA\n",
                  j, length(pam50_pCR[[j]]), sum(is.na(pam50_pCR
[[j]]))))
    } else {
      cat(sprintf("No PAM50 predictions available for study %d\n", j))
    }
  }

  # Compile results
  results <- rbind(
    c(pam_res[1], pam_res[2]),          # PAM50
    c(lr_train_auc, lr_test_auc),       # LR
    c(rf_train_auc, rf_test_auc),       # RF
    c(gbm_train_auc, gbm_test_auc)      # GBM
  )
  rownames(results) <- c("PAM50 benchmark", "Logistic Regression", "Ran
dom Forest", "GBM")
  colnames(results) <- c("Train AUC", "Test AUC")

  return(results)
}


j <- 10
alpha_best <- 0.625
k_best <- 2
# Function to get predictors with sign information
get_predictors_with_sign <- function(XX_cut, groups, k, signs) {
  # XX_cut: G genes × N samples
  # signs: length-G vector of +1/-1
  N  <- ncol(XX_cut)
  P  <- matrix(0, nrow = N, ncol = k, dimnames = list(colnames(XX_cut),
                                                       paste0("Feat",
 1:k)))

  if (length(groups) >= k) {
    # for each of the first k clusters, average signed expression
    for (i in seq_len(k)) {
      gset <- groups[[i]]
      # average (signed) expression of group i, across samples
      P[, i] <- colMeans( XX_cut[gset, , drop=FALSE] * signs[gset] )
    }
  } else {
    # fallback to PCA
    signed_mat <- XX_cut * signs
    Pd <- t(signed_mat)  # samples × genes
    Pd <- apply(Pd, 2, function(v) {
      v[is.na(v)] <- mean(v, na.rm=TRUE); v
    })
    kact <- min(k, ncol(Pd), nrow(Pd) - 1)
    pca  <- tryCatch(prcomp(Pd, center=TRUE, scale.=TRUE), error = func
tion(e) NULL)
    if (!is.null(pca) && kact >= 1) {
      pc_scores <- pca$x[, 1:kact, drop=FALSE]  # samples × kact
      P[, 1:kact] <- pc_scores
    }
  }

  return(P)  # now: rows = samples, cols = features
}


# 1. Combine & standardize across studies
YY <- data.frame(); XX <- NULL
for(i in seq_along(XX_pCR)){
  mat <- XX_pCR[[i]]; resp <- pCR[[i]]
  if(is.null(mat)) next
  ok <- !is.na(resp)
  if(!any(ok)) next
  mat_s <- t(scale(t(mat[,ok,drop=FALSE])))
  if(is.null(XX)) XX <- mat_s
  else {
    common <- intersect(rownames(XX),rownames(mat_s))
    XX <- cbind(XX[common,], mat_s[common,])
  }
  YY <- rbind(YY, data.frame(response=resp[ok], study=i))
}
XX[is.na(XX)] <- 0

# 2. Train/test split
idx_tr <- which(YY$study != j)
idx_te <- which(YY$study == j)

# 3. Feature selection on training only
sel <- threshold_with_sign(XX[,idx_tr,drop=FALSE], YY$response[idx_tr],
 alpha_best)
if(nrow(sel$genes)==0){
  warning("no genes passed; taking top 100 by AUC")
  aucs <- apply(XX[,idx_tr],1,fast_auc,YY$response[idx_tr])
  top  <- order(aucs,decreasing=TRUE)[1:100]
  sel  <- list(
    genes = XX[top,idx_tr,drop=FALSE],
    signs = setNames(ifelse(aucs[top]>=0.5,1,-1), rownames(XX)[top])
  )
```

```r
}
names(sel$signs) <- rownames(sel$genes)

# 4. Cluster & build full-sample predictor matrix
groups      <- corr_cluster(sel$genes, corr_thr=1.0)
genes_sel   <- rownames(sel$genes)
signs_sel   <- sel$signs[genes_sel]
XX_cut_full <- XX[genes_sel,,drop=FALSE]
P           <- get_predictors_with_sign(XX_cut_full, groups, k_best, si
gns_sel)

stopifnot(nrow(P)==nrow(YY))

j <- 10
k_best <- 2

# 1. Split indices
idx_tr <- which(YY$study != j)
idx_te <- which(YY$study == j)

# 2. Fit logistic model on training subset only
lr_mod <- fit_model(
  predictors = P,
  YY_train   = YY$response[idx_tr],
  indexes    = idx_tr,
  k          = k_best,
  lr         = TRUE
)

# 3. Compute AUCs
df_tr <- as.data.frame(P[idx_tr,]); colnames(df_tr) <- paste0("P",1:k_b
est)
df_te <- as.data.frame(P[idx_te,]); colnames(df_te) <- paste0("P",1:k_b
est)

lr_train_auc <- auc_safe(
  predict(lr_mod, newdata = df_tr, type="response"),
  YY$response[idx_tr]
)
lr_test_auc <- auc_safe(
  predict(lr_mod, newdata = df_te, type="response"),
  YY$response[idx_te]
)

# 1. Train/test indices
idx_tr <- which(YY$study != j)
idx_te <- which(YY$study == j)

# 2. Pull out and rename your predictors to P1…Pk
df_tr <- as.data.frame(P[idx_tr, , drop = FALSE])
```

```
df_te <- as.data.frame(P[idx_te, , drop = FALSE])
colnames(df_tr) <- colnames(df_te) <- paste0("P", seq_len(k_best))

# 3. Fit the RF on the training data frame
rf_mod <- randomForest(
  x        = df_tr,
  y        = as.factor(YY$response[idx_tr]),
  ntree    = 500,
  mtry     = floor(sqrt(k_best)),
  maxnodes = 10,          # cap max tree depth
  nodesize = 5            # minimum number of samples per leaf
)

# 4) get predicted probabilities
rf_train_probs <- predict(rf_mod, newdata = df_tr, type = "prob")[,2]
rf_test_probs  <- predict(rf_mod, newdata = df_te, type = "prob")[,2]

# 5) compute AUCs
rf_train_auc <- auc_safe(rf_train_probs, YY$response[idx_tr])
rf_test_auc  <- auc_safe(rf_test_probs,  YY$response[idx_te])

# 6) optionally print
rf_train_auc
```

```
## [1] 0.8195603
```

```
rf_test_auc
```

```
## [1] 0.7149203
```

```
# 1. Extract predictors and response
X_tr <- as.data.frame(P[idx_tr, , drop = FALSE])
X_te <- as.data.frame(P[idx_te, , drop = FALSE])
y_tr <- YY$response[idx_tr]
y_te <- YY$response[idx_te]

# 2. Fit GBM model
gbm_mod <- gbm::gbm(
  formula = y_tr ~ .,
  data = data.frame(y_tr = y_tr, X_tr),
  distribution = "bernoulli",
  n.trees = 100,
  interaction.depth = 3,
  shrinkage = 0.05,
  n.minobsinnode = 5,
  verbose = FALSE
)

# 3. Find best number of trees via internal cross-validation (if using
```

```r
cv.folds)
best_iter <- gbm::gbm.perf(gbm_mod, method = "OOB", plot.it = FALSE)
```

```
## OOB generally underestimates the optimal number of iterations althou
gh predictive performance is reasonably competitive. Using cv_folds>1 w
hen calling gbm usually results in improved predictive performance.
```

```r
# 4. Predict
gbm_train_probs <- predict(gbm_mod, newdata = X_tr, n.trees = best_iter,
 type = "response")
gbm_test_probs  <- predict(gbm_mod, newdata = X_te, n.trees = best_iter,
 type = "response")

# 5. AUC
gbm_train_auc <- auc_safe(gbm_train_probs, y_tr)
gbm_test_auc  <- auc_safe(gbm_test_probs,  y_te)

# print
cat(sprintf("GBM AUCs: train = %.3f, test = %.3f\n", gbm_train_auc, gbm
_test_auc))
```

```
## GBM AUCs: train = 0.730, test = 0.753
```

```r
auc_safe <- function(probs, truth) {
  if (length(unique(na.omit(truth))) != 2 || length(probs) != length(tr
uth)) {
    return(NA_real_)
  }
  tryCatch({
    pred <- ROCR::prediction(probs, truth)
    perf <- ROCR::performance(pred, "auc")
    perf@y.values[[1]]
  }, error = function(e) NA_real_)
}


# results <- run_model_comparison(
#   XX_pCR = XX_pCR,
#   pCR = pCR,
#   pam50_pCR = pam50_pCR,
#   j = 10,
#   alpha_best = 0.625,
#   k_best = 2
# )
# kable(as.data.frame(results), row.names = TRUE)


# Skip aggregation if test set isn't valid
if (length(unique(YY$response[idx_te])) != 2) {
  warning(sprintf("Study %d: test set has only one class — skipping eva
luation", j))
} else {
  pam_res <- benchmark(pam50_pCR, pCR, j)

  results <- rbind(
    c(pam_res[1],     pam_res[2]    ),     # PAM50
    c(lr_train_auc,  lr_test_auc  ),     # LR
    c(rf_train_auc,  rf_test_auc  ),     # RF
    c(gbm_train_auc, gbm_test_auc )      # GBM
  )
  rownames(results) <- c("PAM50 benchmark","Logistic Regression","Rando
m Forest","GBM")
  colnames(results) <- c("Train AUC","Test AUC")

  # Add AUC gap column to compare training vs test
  results <- cbind(results, Gap = results[, "Train AUC"] - results[, "T
est AUC"])

  options(digits = 3)
  kable(as.data.frame(results), row.names = TRUE)
}
```

```
                       Train AUC  Test AUC     Gap
PAM50 benchmark            0.524     0.571  -0.047
Logistic Regression        0.674     0.754  -0.079
Random Forest              0.820     0.715   0.105
GBM                        0.730     0.753  -0.023
```

Discussion

All three models significantly outperformed the Prosigna PAM50 benchmark in both training and test AUC, confirming the effectiveness of our feature engineering pipeline. Even logistic regression, when trained on our selected and compressed features, exceeded the performance of a logistic regression model using raw PAM50 expression. More flexible models like random forest and GBM further improved train performance, though random forest showed signs of overfitting. We see that for random forest, its test AUC dropped by 0.115 relative to training. This suggests that while random forest benefits from strong features, it may require more aggressive regularization or dimensionality reduction.

Overall, logistic regression achieved the highest test AUC (by a small advantage) and appears to be the most reliable model for deployment. It offers both interpretability and competitive performance, making it a promising tool for hospital-level pCR prediction.

Limitations

While our modular approach and leave-one-study-out (LOSO) cross-validation framework enhance the generalizability and reproducibility of results across heterogeneous datasets, several limitations should be acknowledged:

1. Assumption of MKI67 as a Normalization Anchor

To account for between-study technical variation, I used the expression of MKI67 as a normalization reference, treating it as a "housekeeping" gene. MKI67 is indeed associated with cancer cell proliferation and is often used in clinical contexts to evaluate tumor aggressiveness. However, it is not a true housekeeping gene in the classical sense—its expression can vary depending on tumor subtype, treatment status, or even the platform used. This approximation may not fully correct for batch effects, especially in studies with subtle or heterogeneous technical artifacts. A more robust alternative might involve empirically selecting invariant genes or applying established batch correction methods like ComBat or RUV.

2. PCA-based features may not suit all models equally

After filtering for predictive genes, we applied PCA to reduce dimensionality and construct orthogonal predictors. While PCA is effective in summarizing variance and preventing multicollinearity, it is fundamentally a linear transformation. Models like random forest and gradient boosting machines are inherently nonlinear and may not benefit as much from PCA compression, especially if the principal components dilute interpretable nonlinear signals. Tree-based models might instead perform better with raw or grouped gene-level features that retain biological structure.

3. Omission of regularized logistic models

I selected plain logistic regression as our baseline model due to its simplicity and clinical interpretability. However, I did not include regularized variants such as LASSO or Elastic Net, which are specifically designed for high-dimensional settings like gene expression data. Regularization could help enforce sparsity, reduce overfitting, and improve generalizability—especially when the number of predictors exceeds the number of samples. While our PCA-based dimensionality reduction mitigates this concern to some extent, incorporating LASSO in future work may enhance both performance and interpretability.

Despite these trade-offs, our results demonstrate that thoughtful preprocessing and feature construction—coupled with careful model validation—can yield robust and interpretable predictors that exceed clinical benchmarks.