Bayesian Linear Regression

Karl Meng

2024-11-15

Bayesian Linear Regression: Comprehensive Analysis of Principles, Hyperparameter Tuning, and Model Performance

In statistical modeling, linear regression is a fundamental method for predicting continuous outcomes based on one or more predictors. While classical linear regression provides point estimates of parameters, it lacks the capacity to capture uncertainty in these estimates, which is often critical for robust predictions and informed decision-making.

Bayesian linear regression addresses this limitation by incorporating prior beliefs and updating them with observed data via Bayes' theorem. Unlike traditional approaches, it yields posterior distributions for parameters, enabling uncertainty quantification and probabilistic interpretation.

This report is aim to explores Bayesian linear regression with a focus on its application to the Boston Housing Dataset. Key objectives include analyzing feature impacts on housing prices, making uncertainty-aware predictions, and investigating the effects of hyperparameters and training data size on model performance.

Load Packages

The purpose of this step is to load the necessary R packages for conducting Bayesian Linear Regression. If any of these packages are not installed, the code will automatically install them.

```
# Install packages if not already installed
if (!requireNamespace("mvtnorm", quietly = TRUE)) install.packages("mvtnorm")
if (!requireNamespace("dplyr", quietly = TRUE)) install.packages("dplyr")
if (!requireNamespace("ggplot2", quietly = TRUE)) install.packages("ggplot2")
if (!requireNamespace("reshape2", quietly = TRUE)) install.packages("reshape2")
```

Section 1: Bayesian Linear Regression

The purpose of this section is to implement Bayesian Linear Regression using Gibbs Sampling, a Markov Chain Monte Carlo method for drawing samples from the posterior distributions of model parameters. It begins by validating the input dimensions to ensure consistency with the Bayesian model. The process involves precomputing key matrices for efficiency, initializing storage for posterior samples, and setting initial values for the regression coefficients (β) and error variance (σ^2). Gibbs sampling alternates between sampling β from a multivariate normal posterior and σ^2 from an inverse-gamma posterior, capturing the uncertainty in parameter estimates. After discarding the burn-in samples, the function computes summary statistics, including the posterior means and covariance of β and σ^2, providing a comprehensive representation of the posterior distributions. The output is a structured list containing the sampled values and these summary metrics, facilitating further analysis of the Bayesian linear regression model.

```
# Load the necessary package "mvtnorm "
# for multivariate normal distribution sampling
library(mvtnorm)

# Bayesian Linear Regression using Gibbs Sampling
bayesian_linear_regression <- function(X, y, beta0, V0, a0, b0, n_iter, burn_in) {

    #### Step 1: Input Validation ####
    # In this step, we ensure the input dimensions match the mathematical model
    # to prevent dimension mismatches.
    if (ncol(X) != length(beta0)) {
      stop("Dimensions_of_X_and_beta0_do_not_match.")
    }
    if (nrow(V0) != ncol(V0) || ncol(V0) != ncol(X)) {
      stop("Dimensions_of_V0_do_not_match_with_X_or_are_not_square.")
    }

    #### Step 2: Get Dimensions of the Data ####
    # In this step, we retrieve the number of observations (n)
    # and the number of predictors (p).
    n <- nrow(X)  # n represents the number of rows in the data (observations)
    p <- ncol(X)  # p represents the number of columns in the data (predictors)

    #### Step 3: Precompute Matrices ####
    # In this step, we precompute matrices used repeatedly in the Gibbs sampling
    # process to save computation time.
    XtX <- crossprod(X)  # Compute the predictor covariance matrix
    Xty <- crossprod(X, y)  # Captures relationship between predictors and response
    V0_inv <- solve(V0)  # Inverse of the prior covariance matrix for beta
    beta0_V0_inv <- V0_inv %*% beta0 # Combines prior mean and covariance for efficient calculations.
```

```r
#### Step 4: Initialize Storage for Posterior Samples ####
# In this step we create matrices to store the sampled beta and
# sigma^2 values for later analysis.
beta_samples <- matrix(NA, nrow = n_iter, ncol = p)  # To store sampled beta values
sigma2_samples <- numeric(n_iter)  # To store sampled sigma^2 values

#### Step 5: Set Initial Values for Beta and Sigma^2 ####
# In this step, we start the Gibbs sampling process with
# reasonable initial values.
beta_current <- beta0  # Initialize beta as the prior mean
sigma2_current <- 1    # Initialize sigma^2 with a default value

#### Step 6: Gibbs Sampling Iterations ####
# In this step, we alternate between sampling beta and sigma^2 from their
# conditional posterior distributions.
for (iter in 1:n_iter) {

  ## Sample beta | sigma^2, y ##
  # Using conditional posterior distribution of beta given sigma^2 and the data

  # Compute the posterior precision matrix (inverse of posterior covariance).
  V_N_inv <- V0_inv + XtX / sigma2_current
  # Compute the posterior covariance matrix for beta.
  V_N <- solve(V_N_inv)
  # Compute the posterior mean for beta.
  beta_N <- V_N %*% (beta0_V0_inv + Xty / sigma2_current)
  # Sample beta from the posterior distribution (multivariate normal).
  beta_current <- as.numeric(rmvnorm(1, mean = beta_N, sigma = V_N))

  # Notice that the posterior distribution of beta reflects both the prior
  # belief (beta0, V0) and the data information (XtX, Xty). Sampling from
  # this distribution captures the uncertainty in beta estimation.

  ## Sample sigma^2 | beta, y ##
  # Using conditional posterior distribution of sigma^2 given beta and the data

  # Compute residuals based on current beta.
  residual <- y - X %*% beta_current # Compute residuals based on current beta
  # Compute the shape parameter for the posterior inverse-gamma distribution.
  a_N <- a0 + n / 2  # Shape parameter
  b_N <- b0 + 0.5 * crossprod(residual) +
       0.5 * t(beta_current - beta0) %*% V0_inv %*% (beta_current - beta0)
  # Sample sigma^2 from the posterior distribution (inverse-gamma).
  sigma2_current <- 1 / rgamma(1, shape = a_N, rate = b_N)

  # Notice that sigma^2 represents the variance of the residuals.
  # Its posterior combines prior uncertainty with information from
  # the observed residuals and beta uncertainty.

  ## Store the sampled values for beta and sigma^2 ##
  # Save the current samples of beta and sigma^2 for later analysis.
  beta_samples[iter, ] <- beta_current
  sigma2_samples[iter] <- sigma2_current
}

#### Step 7: Discard Burn-in Samples ####
# In this step, we remove initial samples to allow the Markov chain to
# reach its stationary distribution.
beta_samples <- beta_samples[-(1:burn_in), , drop = FALSE]
sigma2_samples <- sigma2_samples[-(1:burn_in)]

#### Step 8: Compute Summary Statistics ####
# In this step, we summarize the posterior distributions by computing
# means and covariance.
beta_mean <- colMeans(beta_samples)  # Posterior mean of beta samples
sigma2_mean <- mean(sigma2_samples)  # Posterior mean of sigma^2 samples
beta_cov <- cov(beta_samples)  # Posterior Covariance matrix of beta samples

#### Step 9: Return Results ####
# Lastly, we return the sampled values and summary statistics as a list.

return(list(
  beta_samples = beta_samples,  # All sampled beta values
  sigma2_samples = sigma2_samples,  # All sampled sigma^2 values
```

```
    beta_mean = beta_mean,  # Mean of the posterior beta samples
    sigma2_mean = sigma2_mean, # Mean of the posterior sigma^2 samples
    beta_cov = beta_cov  # Covariance matrix of the posterior beta samples
  ))
}
```

The purpose of this section is to implement a prediction function for Bayesian Linear Regression, leveraging posterior samples of regression coefficients (β) and residual variance (σ^2) to generate predictions for new data points. It begins by determining the dimensions of the posterior samples and the new design matrix (X_{new}), ensuring proper alignment. Predictions are generated iteratively for each posterior sample by combining sampled coefficients and variance with the design matrix, capturing the uncertainty in predictions. The function then calculates summary statistics, including the mean and 95% credible intervals, to provide both point estimates and uncertainty quantification for each prediction. The output is a data frame containing the predictive mean and credible intervals, offering a clear representation of the posterior predictive distribution.

```
# Prediction Function for Bayesian Linear Regression
predict_bayesian_lr <- function(beta_samples, sigma2_samples, X_new) {
  # Inputs are as following:
  # - beta_samples: Matrix of posterior samples of beta coefficients (each row corresponds to a sampled beta vect
or)
  # - sigma2_samples: Vector of posterior samples of sigma^2 (residual variance)
  # - X_new: Design matrix for new data points (each row is a new data point)

  #### Step 1: Get Dimensions of Inputs ####
  # In this step, we determine the number of posterior samples and the
  # number of new data points.
  n_samples <- nrow(beta_samples) # Number of posterior samples
  n_new <- nrow(X_new) # Number of new data points to predict

  #### Step 2: Initialize Storage for Predictions ####
  # In this step, we create a matrix to store the predicted values for
  # all posterior samples.
  # Each row corresponds to predictions for one posterior sample,
  # and each column corresponds to a new data point.
  y_pred_samples <- matrix(NA, nrow = n_samples, ncol = n_new)

  #### Step 3: Generate Predictive Values ####
  # For each posterior sample of beta and sigma^2, we generate predicted values.
  for (i in 1:n_samples) {
    # Extract the i-th posterior sample of beta coefficients.
    beta_i <- beta_samples[i, ]
    # Extract the i-th posterior sample of sigma^2 (residual variance).
    sigma2_i <- sigma2_samples[i]

    # Compute predictions for new data points based on the sampled parameters.
    # Predictions are calculated as the linear model plus random noise:
    y_pred_samples[i, ] <- X_new %*% beta_i + rnorm(n_new, mean = 0, sd = sqrt(sigma2_i))

    # Notice that this step generates predictive values that
    # incorporate uncertainty from both the posterior distribution of
    # beta (model coefficients) and sigma^2 (residual variance).
    # By adding random noise sampled from N(0, sigma2_i),
    # we reflect the uncertainty in the predictions.
  }

  #### Step 4: Compute Predictive Summary Statistics ####
  # In this step, we calculate the mean and 95% credible interval
  # for the predictions.

  # Compute the posterior predictive mean for each new data point.
  pred_mean <- colMeans(y_pred_samples)  # Mean across all posterior samples

  # Compute the 2.5th percentile (lower bound of 95% credible interval)
  # for each prediction.
  pred_lower <- apply(y_pred_samples, 2, quantile, probs = 0.025)

  # Compute the 97.5th percentile (upper bound of 95% credible interval)
  # for each prediction.
  pred_upper <- apply(y_pred_samples, 2, quantile, probs = 0.975)

  # Notice that these summary statistics provide an interpretable representation of
  # the posterior predictive distribution for each new data point,
  # including both point estimates and uncertainty.
```

```
  #### Step 5: Return Predictions ####
  # In this step, we return a data frame containing the predictive mean and
  # 95% credible intervals for each new data point.
  return(data.frame(
    pred_mean = pred_mean,
    pred_lower = pred_lower,
    pred_upper = pred_upper
  ))
 }
```

Section 2: Data Loading and Preprocessing

The purpose of this section is to prepare the Boston Housing dataset for Bayesian Linear Regression. The dataset is loaded from a CSV file and previewed to inspect its structure, data types, and content, ensuring it is correctly imported. Missing values are identified and addressed using mean imputation, where each missing value is replaced with the mean of its respective column, ensuring a complete and clean dataset. After cleaning, the data is split into training, validation, and test sets with proportions of 80%, 10%, and 10%, respectively, to enable robust model training and evaluation.

The Boston Housing dataset is selected because it provides a well-known benchmark for regression tasks, with features that capture various aspects of housing data, such as socioeconomic and environmental factors. This dataset's size and feature diversity make it ideal for demonstrating Bayesian Linear Regression, as it allows exploration of uncertainty in parameter estimates and predictions, which are central to Bayesian approaches.

```
# Load the necessary package "dplyr"
# for data manipulation functions.
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
#### Step 1: Load the Dataset ####
# In this step, we load the Boston Housing dataset from a CSV file.
Boston <- read.csv("BostonHousing_data.csv")

#### Step 2: Check for Missing Values ####
# In this step, we calculate the total number of missing values in the dataset.
missing_count <- sum(is.na(Boston))

# Check if there are any missing values and print the result.
if (missing_count == 0) {
  cat("The dataset has no missing values.\n")
} else {
  cat("The dataset has", missing_count, "missing values.\n")
}
```

```
## The dataset has 5 missing values.
```

```
# Notice that identifying missing values is important before proceeding
# with data analysis, which informs us whether we need to handle missing data.

#### Step 4: Impute Missing Values Using Mean Imputation ####
# Based on the outcome above, if there are missing values,
# use mean imputation to fill them.
# For each column, replace NA values with the mean of that column.
Boston <- Boston %>% mutate_all(~ ifelse(is.na(.), mean(., na.rm = TRUE), .))

# Notice that we use the method mean imputation to replace missing values
# with the mean of the non-missing values in the column.
# This method maintains the dataset size and uses all available data for modeling.
# It assumes that data are missing at random and that
# the mean is a reasonable estimate.
```

```r
#### Step 5: Inspect the Data ####
# In this step, we print the first few rows of the dataset to examine its structure
# and ensure it is ready for further analysis.
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio      b lstat
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
## 4 33.4
## 5 36.2
## 6 28.7
```

```r
#### Step 1: Recalculate Missing Values After Imputation ####
# In this step, we confirm that all missing values have been handled.
# After performing mean imputation, it's important to verify that
# there are no remaining missing values in the dataset.
issing_count <- sum(is.na(Boston))
cat("After cleaning, the dataset has", missing_count, "missing values.\n")
```

```
## After cleaning, the dataset has 5 missing values.
```

```r
#### Step 2: Determine the Total Number of Observations ####
# In this step, we obtain the total number of data points for splitting.
# Get the total number of observations (rows) in the dataset.
n <- nrow(Boston)

#### Step 3: Define Proportions for Data Splitting ####
# In this step, we set the suitable proportions for each dataset.
train_prop <- 0.8       # 80% for training
validation_prop <- 0.1  # 10% for validation
test_prop <- 0.1        # 10% for testing

# Ensure the proportions sum to 1.
total_prop <- train_prop + validation_prop + test_prop
if (total_prop != 1) {
  stop("The sum of the split proportions must equal 1.")
}

# Set a Random Seed for Reproducibility.
set.seed(123)

#### Step 4: Calculate the Number of Observations for Each Set ####
# In this step, we determine the exact number of observations for training, validation, and test sets:

# Calculate the number of observations for the training set.
train_size <- floor(train_prop * n)
# Calculate the number of observations for the validation set.
validation_size <- floor(validation_prop * n)
# Assign the remaining observations to the test set.
test_size <- n - train_size - validation_size

#### Step 5: Shuffle the Data Indices ####
# In this step, we randomize the order of the data before splitting to
# ensure that the data is randomly distributed among the sets,
# preventing any order bias.
shuffled_indices <- sample(n) # Create a vector of shuffled row indices.

#### Step 6: Split the Indices into Training, Validation, and Test Sets ####
# In this step, we assign data points to each set based on the calculated sizes.
# Using indices allows us to subset the dataset without altering the original data.

# Assign indices for the training set.
train_indices <- shuffled_indices[1:train_size]
# Assign indices for the validation set.
```

```r
validation_indices <- shuffled_indices[(train_size + 1):(train_size + validation_size)]
# Assign indices for the test set.
test_indices <- shuffled_indices[(train_size + validation_size + 1):n]

#### Step 8: Create the Training, Validation, and Test Sets ####
# In this step, we create mutually exclusive and collectively exhaustive
# subsets of the data.

# Create the training set.
train_data <- Boston[train_indices, ]
# Create the validation set.
validation_data <- Boston[validation_indices, ]
# Create the test set.
test_data <- Boston[test_indices, ]

# Print the sizes of each set
cat("Training set size:", nrow(train_data), "\n")
```

```
## Training set size: 404
```

```r
cat("Validation set size:", nrow(validation_data), "\n")
```

```
## Validation set size: 50
```

```r
cat("Test set size:", nrow(test_data), "\n")
```

```
## Test set size: 52
```

The purpose of this section is to perform Bayesian Linear Regression to estimate the impact of each feature on the response variable. The training, validation, and test datasets are constructed by separating predictors and the response variable while adding an intercept term. Prior distributions are defined with specified mean, covariance, and hyperparameters for the coefficients and residual variance. Gibbs Sampling is then executed to draw posterior samples, and the posterior means and 95% credible intervals for each regression coefficient are computed to quantify their impact. These results are summarized in a table for clear interpretation of the feature effects.

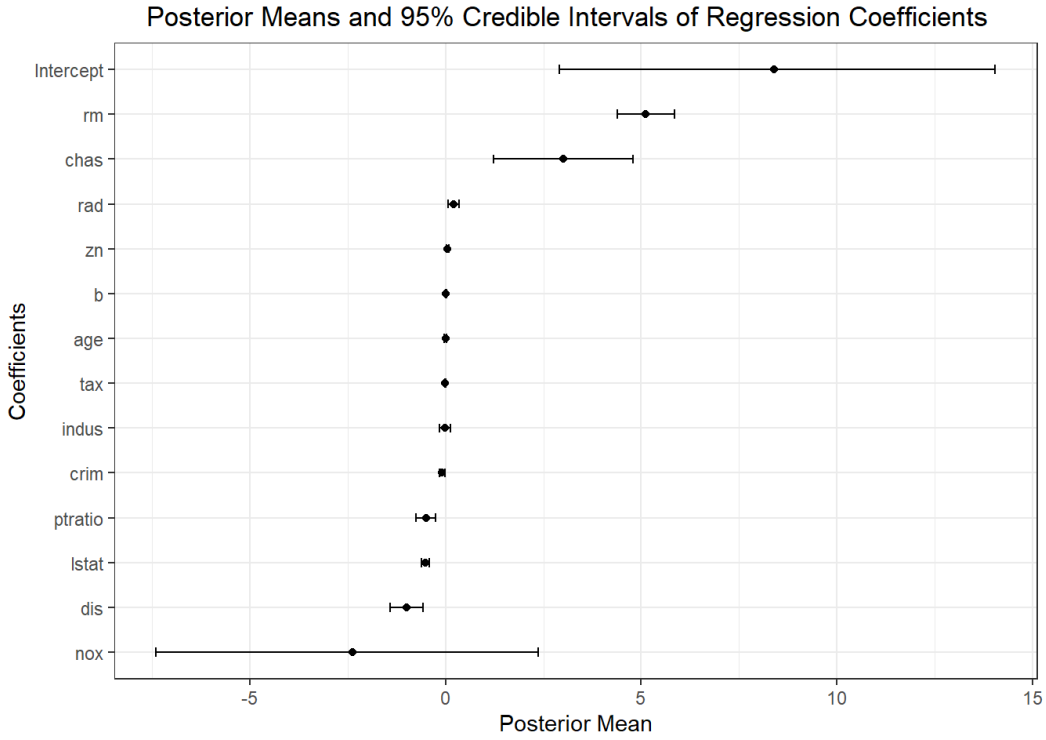
```r
# Load the necessary package "dplyr" for data manipulation functions .
library (dplyr)

#### Step 1: Prepare Training Data, Validation Data and Test Data ####

# Select all columns except 'medv' as predictors and convert to matrix format.
X_train <- train_data %>% dplyr::select(-medv) # exclude 'medv' since it's the target variable.
X_train <- as.matrix(X_train)
# Set the response variable as the 'medv' column.
y_train <- train_data$medv
# Add an intercept term to the predictors allows the model to learn a bias term.
X_train <- cbind(Intercept = 1, X_train)

# Select predictors and convert to matrix format.
X_validation <- validation_data %>% dplyr::select(-medv)
X_validation <- as.matrix(X_validation)
# Add an intercept term.
X_validation <- cbind(Intercept = 1, X_validation)
# Set the response variable.
y_validation <- validation_data$medv

# Select predictors and convert to matrix format.
X_test <- test_data %>% dplyr::select(-medv)
X_test <- as.matrix(X_test)
# Add an intercept term.
X_test <- cbind(Intercept = 1, X_test)
# Set the response variable.
y_test <- test_data$medv

#### Step 2: Set Prior Parameters for Bayesian Linear Regression ####
# In this step, we define the prior distributions for the model parameters.
# Number of predictors (including the intercept)
p <- ncol(X_train)
# Prior mean vector for regression coefficients (initialized to zeros)
beta0 <- rep(0, p)
# Prior variance determined from previous analysis.
```

```r
tau2 <- 10
# Prior covariance matrix (scaled identity matrix).
V0 <- diag(tau2, p)
# Prior shape parameter for the inverse-gamma distribution
a0 <- 0.01
# Prior scale parameter for the inverse-gamma distribution
b0 <- 0.01

#### Step 3: Specify Gibbs Sampling Parameters ####
# In this step, we define the sampling parameters for the Gibbs sampler.
# Total number of iterations for Gibbs sampling.
n_iter <- 5000
# Number of initial samples to discard (burn-in period).
burn_in <- 1000

# Set a Random Seed for Reproducibility
set.seed(123)

#### Step 4: Run Bayesian Linear Regression Using Gibbs Sampling ####
# In this step, we Obtain posterior samples of the model parameters.
bayesian_results <- bayesian_linear_regression(
  X_train, y_train, beta0, V0, a0, b0, n_iter, burn_in
)

#### Step 5: Extract Posterior Samples ####
# In this step, we extract posterior samples for beta coefficients and variance.
beta_samples <- bayesian_results$beta_samples
sigma2_samples <- bayesian_results$sigma2_samples

#### Step 6: Estimate the Impact of Each Feature ####
# In this step, we summarize the posterior distributions to # interpret
# feature effects.

# Calculate the posterior means of beta coefficients (point estimates).
beta_posterior_mean <- colMeans(beta_samples)

# Calculate 95% credible intervals for beta coefficients.
beta_ci_lower <- apply(beta_samples, 2, quantile, probs = 0.025)  # 2.5th percentile
beta_ci_upper <- apply(beta_samples, 2, quantile, probs = 0.975)  # 97.5th percentile

#### Step 7: Create a Summary Table of Coefficients ####
# In this step, we create a summary table with coefficient names, means,
# and credible intervals.
beta_summary <- data.frame(
  Coefficient = colnames(X_train),  # Names of the coefficients (including intercept).
  Mean = beta_posterior_mean,       # Posterior mean estimates
  Lower = beta_ci_lower,            # Lower bound of 95% credible interval
  Upper = beta_ci_upper             # Upper bound of 95% credible interval
)

# Finally, display the summary table for further analysis
print(beta_summary)
```

```
##    Coefficient         Mean        Lower         Upper
## 1    Intercept  8.384326796  2.914380257 14.0464547667
## 2         crim -0.090718917 -0.161440459 -0.0204730404
## 3           zn  0.052041992  0.021322925  0.0843211767
## 4        indus -0.015550637 -0.155130792  0.1288457513
## 5         chas  2.998442434  1.228953963  4.7994189881
## 6          nox -2.381771602 -7.400037089  2.3641946108
## 7           rm  5.100695131  4.380822042  5.8442944353
## 8          age  0.003622849 -0.025929324  0.0330945615
## 9          dis -0.990619143 -1.415026574 -0.5669501994
## 10         rad  0.200731949  0.061090413  0.3489586508
## 11         tax -0.009710066 -0.018727149 -0.0009417251
## 12     ptratio -0.502471715 -0.762722306 -0.2478851427
## 13           b  0.012231944  0.006054213  0.0183564001
## 14       lstat -0.516774558 -0.622575244 -0.4081941924
```

```r
# Load the package 'ggplot2'
library(ggplot2)

# Plot the posterior means and credible intervals of regression coefficients
```

```
posterior_plot <- ggplot(beta_summary, aes(x = reorder(Coefficient, Mean), y = Mean)) +
  geom_point() +
  geom_errorbar(aes(ymin = Lower, ymax = Upper), width = 0.2) +
  coord_flip() +
  theme_bw() +
  ggtitle("Posterior Means and 95% Credible Intervals of Regression Coefficients") +
  xlab("Coefficients") +
  ylab("Posterior Mean") +
  theme(plot.title = element_text(hjust = 0.4))

# Print the posterior_plot
print(posterior_plot)
```



The purpose of this section is to apply the Bayesian Linear Regression prediction function to generate predictions for the test set, combining them with actual values for evaluation. The predicted mean values and 95% credible intervals are computed for each test data point to reflect uncertainty in the predictions. Residuals are calculated to measure prediction errors, providing insights into the model's performance and areas for potential refinement.These results are consolidated into a data frame alongside the actual test values, enabling a comparison of predictions to ground truth. Additionally,

```
#Set a Random Seed for Reproducibility ####
set.seed(123)

#### Step 1: Define Prediction Function for Bayesian Linear Regression ####
# In this step, we create a function to generate predictions using
# posterior samples.
predict_bayesian_lr <- function(beta_samples, sigma2_samples, X_new) {

  # Determine the number of posterior samples and new data points.
  n_samples <- nrow(beta_samples)  # Number of posterior samples
  n_new <- nrow(X_new)             # Number of new data points to predict

  # Create a matrix to store predicted values for each posterior sample.
  y_pred_samples <- matrix(NA, nrow = n_samples, ncol = n_new)
  # Each row corresponds to predictions from one posterior sample.

  # Loop through each posterior sample to generate predictions.
  for (i in 1:n_samples) {
    # Extract the i-th posterior sample of beta coefficients.
    beta_i <- beta_samples[i, ]
    # Extract the i-th posterior sample of sigma^2 (residual variance).
    sigma2_i <- sigma2_samples[i]

    # Generate predictive values for the new data points.
    y_pred_samples[i, ] <- X_new %*% beta_i + rnorm(n_new, mean = 0, sd = sqrt(sigma2_i))

    # Notice that the prediction is based on the linear model with added random
    # noise sampled from N(0, sigma^2), reflecting the uncertainty in the model.
```

```r
  }

  # Calculate the mean and 95% credible intervals for the predictions.
  pred_mean <- colMeans(y_pred_samples)  # Mean predicted value across posterior samples
  pred_lower <- apply(y_pred_samples, 2, quantile, probs = 0.025)  # Lower bound (2.5th percentile)
  pred_upper <- apply(y_pred_samples, 2, quantile, probs = 0.975)  # Upper bound (97.5th percentile)

  # Return a data frame with the predictive mean and credible intervals.
  return(data.frame(pred_mean = pred_mean, pred_lower = pred_lower, pred_upper = pred_upper))
}

#### Step 3: Make Predictions on the Test Set ####
# In this step, we use the prediction function to generate predictions
# for the test data.
predictions <- predict_bayesian_lr(beta_samples, sigma2_samples, X_test)

#### Step 4: Combine Predictions with Actual Test Set Values ####
# In this step, we create a data frame to compare predicted values
# with actual values.
prediction_results <- data.frame(
  Actual = y_test,                    # Actual values from the test set
  Predicted = predictions$pred_mean,   # Predicted mean values
  Lower = predictions$pred_lower,      # Lower bound of 95% credible interval
  Upper = predictions$pred_upper       # Upper bound of 95% credible interval
)

#### Step 5: Calculate Residuals ####
# In this step, we compute the differences between actual and predicted values
# to assess prediction errors.
# Residuals provide insight into the prediction errors
prediction_results$Residuals <- prediction_results$Actual - prediction_results$Predicted
```

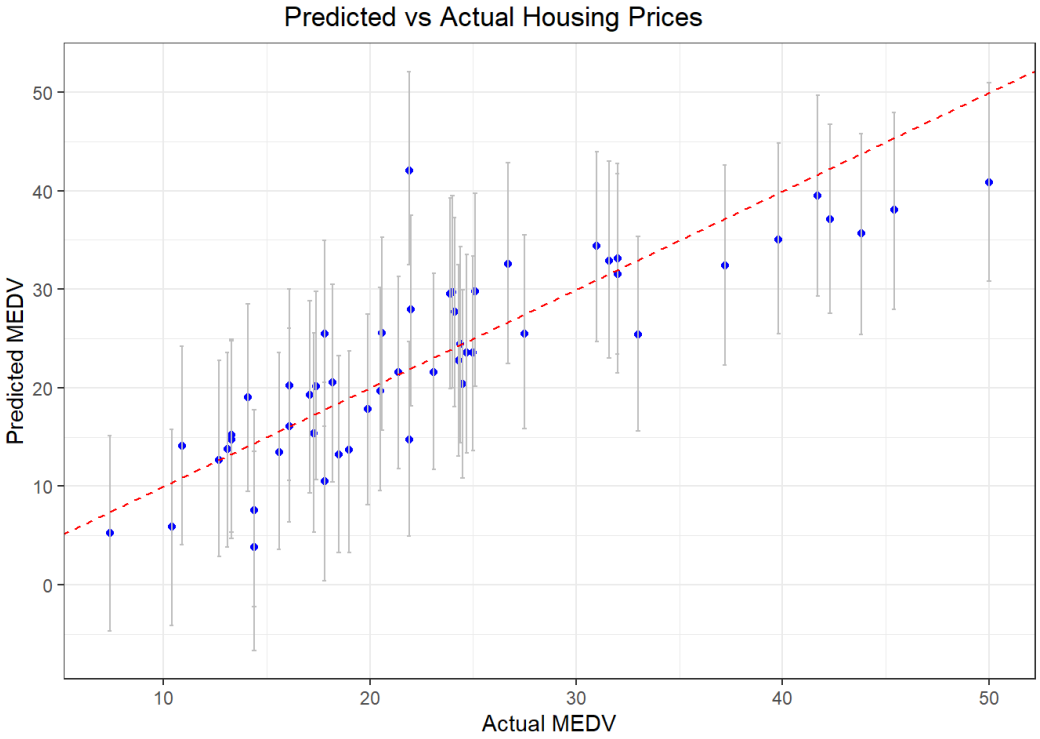
```r
# Load necessary package 'dplyr'
library(dplyr)

# Identify significant predictors
significant_predictors <- beta_summary %>%
  filter(Lower > 0 | Upper < 0)

cat("\nSignificant predictors at 95% credible interval:\n")
```

```
##
## Significant predictors at 95% credible interval:
```

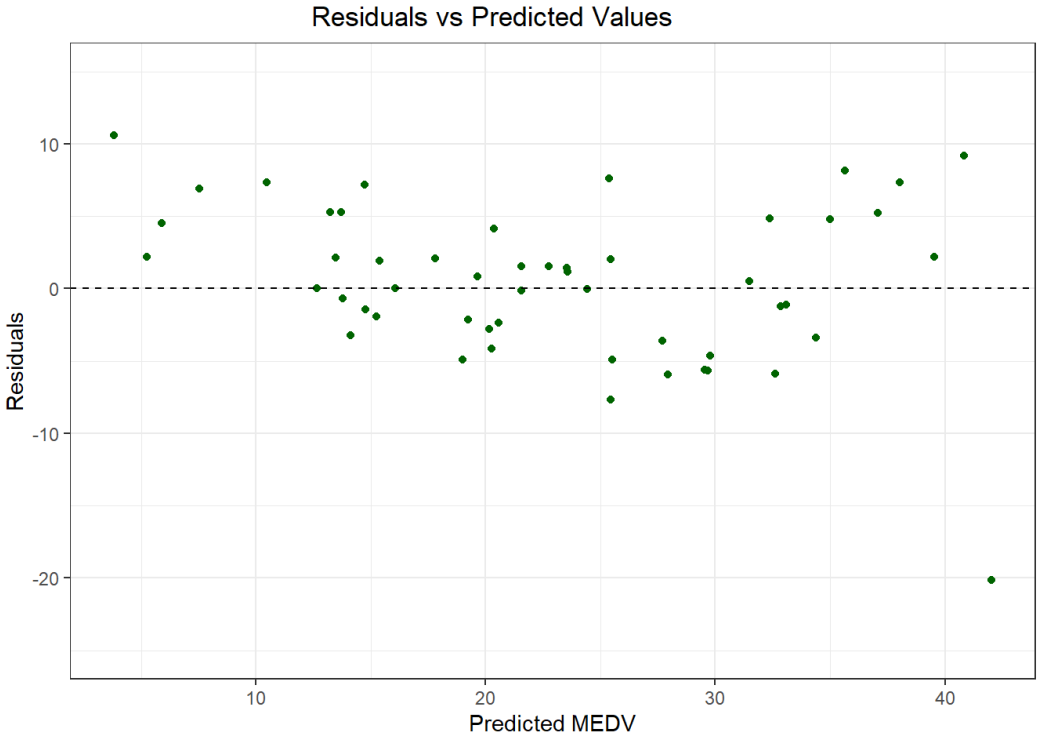
```r
print(significant_predictors$Coefficient)
```

```
##  [1] "Intercept" "crim"      "zn"        "chas"      "rm"        "dis"
##  [7] "rad"       "tax"       "ptratio"   "b"         "lstat"
```

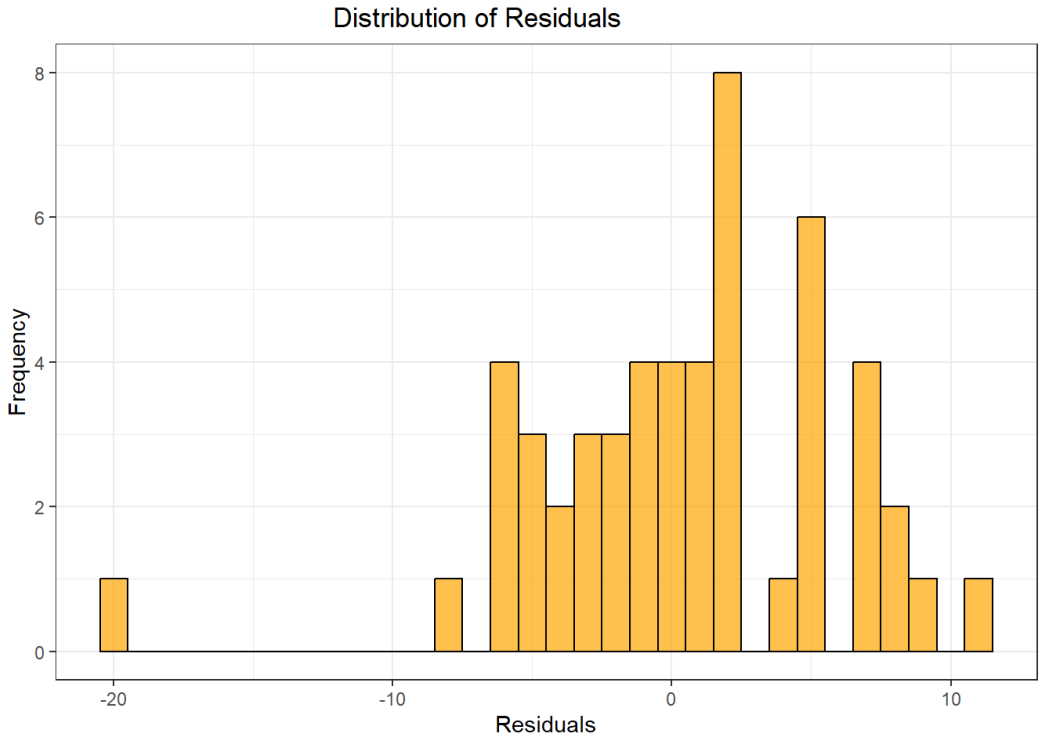
```r
# Load the package 'ggplot2'
library(ggplot2)

# Plot actual vs predicted values with credible intervals
plot_actual_vs_predicted <- ggplot(prediction_results, aes(x = Actual, y = Predicted)) +
  geom_point(color = "blue") +
  geom_errorbar(aes(ymin = Lower, ymax = Upper), width = 0.2, color = "gray") +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  theme_bw() +
  ggtitle("Predicted vs Actual Housing Prices") +
  xlab("Actual MEDV") +
  ylab("Predicted MEDV")+
  theme(
    plot.title = element_text(hjust = 0.4)
  )

print(plot_actual_vs_predicted)
```




```r
# Load the package 'ggplot2'
library(ggplot2)

# Plot residuals vs predicted values with adjusted y-axis
plot_residuals_vs_predicted <- ggplot(prediction_results, aes(x = Predicted, y = Residuals)) +
  geom_point(color = "darkgreen") +
  geom_hline(yintercept = 0, linetype = "dashed") +
  theme_bw() +
  ggtitle("Residuals vs Predicted Values") +
  xlab("Predicted MEDV") +
  ylab("Residuals") +
  ylim(-25, 15) +
  theme(
    plot.title = element_text(hjust = 0.4)
  )

print(plot_residuals_vs_predicted)
```



```r
# Load the package 'ggplot2'
library(ggplot2)

# Histogram of residuals with adjusted x-axis
plot_residuals_histogram <- ggplot(prediction_results, aes(x = Residuals)) +
  geom_histogram(binwidth = 1, fill = "orange", color = "black", alpha = 0.7) +
  theme_bw() +
  ggtitle("Distribution of Residuals") +
  xlab("Residuals") +
  ylab("Frequency") +
  theme(
    plot.title = element_text(hjust = 0.4)
  )

print(plot_residuals_histogram)
```



Section 3: Performance Evaluation of the Bayesian Linear Regression Model

The purpose of this section is to evaluate the performance of a Bayesian Linear Regression model on the Boston Housing dataset by calculating the RMSE for training, validation, and test sets across varying training set proportions. Firstly, we define the feature matrix and target vector, adding an intercept term for the regression model. Training set proportions are specified, and the dataset is randomly shuffled and split into training, validation, and test subsets. For each training set proportion, Bayesian Linear Regression is performed with specified prior parameters, and the posterior mean of coefficients is used to make predictions on all subsets. The RMSE is computed for each subset to assess model performance. Finally, we combine the RMSE results a data frame.

```r
# Load the necessary package "dplyr" package for data manipulation functions.
library(dplyr)

#### Step 1: Prepare the Feature Matrix and Target Vector ####
# In this step, we set up the predictors (X) and response variable (y)
# for the entire dataset.

# Select all columns except "medv" as predictors and convert to matrix format.
X <- as.matrix(Boston %>% dplyr::select(-medv))  # "medv" is the target variable
# Set the response variable as the "medv" column.
y <- Boston$medv  # Target variable (median house values)
# Add an intercept term to the feature matrix.
X <- cbind(Intercept = 1, X)  # Adds a column of ones for the intercept term

#### Step 2: Define Training Set Proportions ####
# In this step, we specify different proportions of data to be used for training.
train_proportions <- c(0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8)  # Training set sizes

#### Step 3: Initialize Vectors to Store RMSE Values ####
# In this step, we prepare storage for RMSE values computed at
# each training set size.
```

```r
rmse_train <- numeric(length(train_proportions))  # Stores RMSE for training data
rmse_val <- numeric(length(train_proportions))    # Stores RMSE for validation data
rmse_test <- numeric(length(train_proportions))   # Stores RMSE for test data

# Set a Random Seed for Reproducibility
set.seed(123)

# Randomly shuffle indices to randomize data splitting.
n_total <- nrow(X)  # Total number of observations in the dataset.
indices <- sample(1:n_total)  # Shuffled indices.

#### Step 4: Loop Over Different Training Set Sizes ####
# In this step, we evaluate model performance for each training set size.
for (i in seq_along(train_proportions)) {

  # Calculate the number of samples for training, validation, and test sets.
  n_train <- floor(train_proportions[i] * n_total)  # Number of training samples
  n_remaining <- n_total - n_train  # Remaining samples for validation and testing
  n_val <- floor(n_remaining / 2)  # Half of remaining samples for validation set
  n_test <- n_remaining - n_val  # Remaining samples for test set

  # Assign indices to each set based on calculated sizes.
  train_indices <- indices[1:n_train]  # Indices for training set
  val_indices <- indices[(n_train + 1):(n_train + n_val)]  # Indices for validation set
  test_indices <- indices[(n_train + n_val + 1):n_total]  # Indices for test set

  # Subset the data using the assigned indices.
  # Training set.
  X_train <- X[train_indices, ]  # Predictor matrix for training set
  y_train <- y[train_indices]    # Target vector for training set

  # Validation set.
  X_val <- X[val_indices, ]  # Predictor matrix for validation set
  y_val <- y[val_indices]    # Target vector for validation set

  # Test set.
  X_test <- X[test_indices, ]  # Predictor matrix for test set
  y_test <- y[test_indices]    # Target vector for test set

  # Define prior distributions for model parameters.
  beta0 <- rep(0, ncol(X))    # Prior mean vector for regression coefficients
  V0 <- diag(1000, ncol(X))   # Prior covariance matrix (scaled identity matrix)
  a0 <- 0.01                  # Shape parameter for inverse-gamma prior on variance
  b0 <- 0.01                  # Scale parameter for inverse-gamma prior on variance

  # Set parameters for the Gibbs sampling algorithm.
  n_iter <- 5000  # Total number of iterations for Gibbs sampling
  burn_in <- 1000  # Number of initial samples to discard (burn-in period)

  # Fit the model and obtain posterior samples.
  results <- bayesian_linear_regression(
    X_train, y_train, beta0, V0, a0, b0,
    n_iter = n_iter, burn_in = burn_in
  )

  # Estimate regression coefficients using posterior means.
  beta_posterior_mean <- colMeans(results$beta_samples)  # Posterior mean estimates

  # Evaluate model performance on training, validation, and test sets.
  # Predictions on training data.
  y_pred_train <- X_train %*% beta_posterior_mean  # Predicted values for training set
  rmse_train[i] <- sqrt(mean((y_train - y_pred_train)^2))  # RMSE for training data

  # Predictions on validation data.
  y_pred_val <- X_val %*% beta_posterior_mean # Predicted values for validation set
  rmse_val[i] <- sqrt(mean((y_val - y_pred_val)^2))  # RMSE for validation data

  # Predictions on test data.
  y_pred_test <- X_test %*% beta_posterior_mean  # Predicted values for test set
  rmse_test[i] <- sqrt(mean((y_test - y_pred_test)^2))  # RMSE for test data

}

#### Step 5: Combine RMSE Results into a Data Frame ####
# In this step, we organize RMSE results for further analysis and plotting.
```

```r
rmse_results <- data.frame(
  TrainingProportion = train_proportions * 100,  # Convert proportions to percentages.
  RMSE_Train = rmse_train,         # RMSE values for training data
  RMSE_Validation = rmse_val,      # RMSE values for validation data
  RMSE_Test = rmse_test            # RMSE values for test data
)

# Print RMSE results
print(rmse_results)  # Display the RMSE results for different training proportions
```

```
##   TrainingProportion RMSE_Train RMSE_Validation RMSE_Test
## 1                 20   4.365591        4.880260  5.494768
## 2                 30   4.224370        4.840428  5.540261
## 3                 40   4.275963        5.472803  4.924471
## 4                 50   4.547394        4.967670  4.949412
## 5                 60   4.422966        5.488652  4.951604
## 6                 70   4.699030        4.577155  5.027353
## 7                 80   4.655953        4.442790  5.245174
```

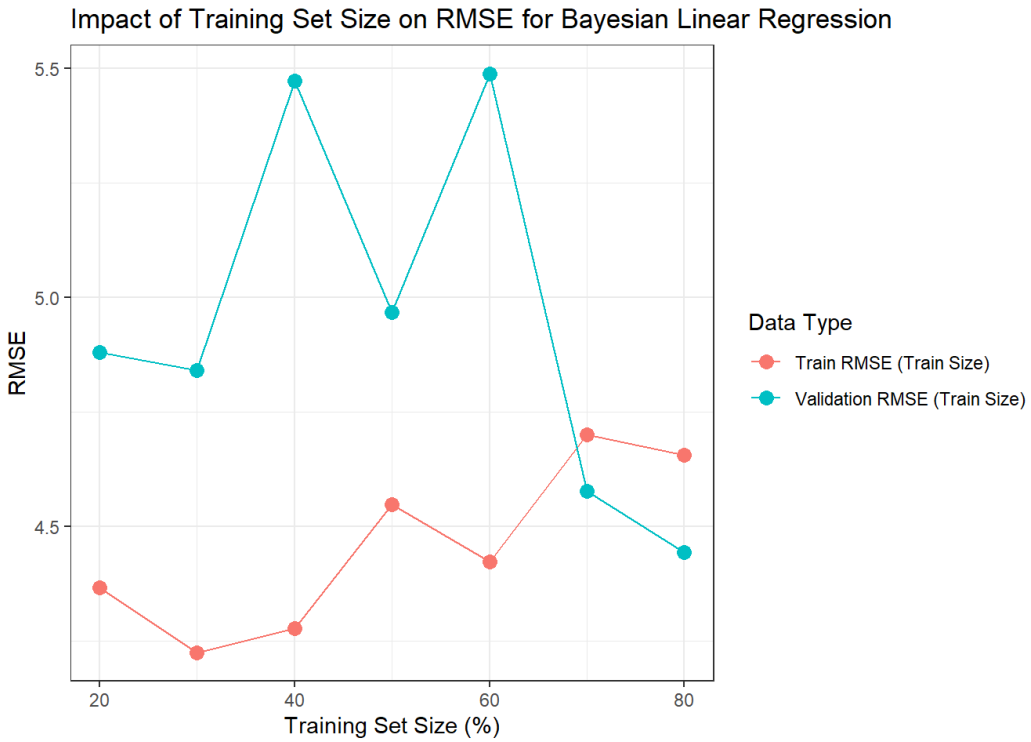
```r
# Find the optimal training set size
optimal_index <- which.min(rmse_val)  # Index of minimum RMSE in validation set
optimal_train_size <- train_proportions[optimal_index] * 100  # Convert to percentage
optimal_rmse <- rmse_val[optimal_index]  # Corresponding validation RMSE

# Output the results
cat("Optimal Training Size (%):", optimal_train_size, "\n")
```

```
## Optimal Training Size (%): 80
```

```r
cat("Validation RMSE at Optimal Training Size:", optimal_rmse, "\n")
```

```
## Validation RMSE at Optimal Training Size: 4.44279
```

```r
# Load the package 'ggplot2'
library(ggplot2)

# Create RMSE data for plotting with more descriptive data types
rmse_data <- data.frame(
  TrainingSetSize = rep(train_proportions * 100, 2),
  RMSE = c(rmse_train, rmse_val),
  Type = rep(c("Train RMSE (Train Size)", "Validation RMSE (Train Size)"), each = length(train_proportions))
)

# Plot RMSE for Training and Validation
rmse_plot <- ggplot(rmse_data, aes(x = TrainingSetSize, y = RMSE, color = Type)) +
  geom_point(size = 3) +
  geom_line() +
  labs(
    title = "Impact of Training Set Size on RMSE for Bayesian Linear Regression",
    x = "Training Set Size (%)",
    y = "RMSE",
    color = "Data Type"
  ) +
  theme_bw()

# Print the plot
print(rmse_plot)
```



Section 4: Exploring Model Performance by Varying a Hyperparameter

The purpose of this section is to evaluate the impact of different prior variances (τ^2) on the performance of the Bayesian Linear Regression model by computing RMSE for both training and validation datasets. The feature matrix and target vector are first defined, with an intercept term added to the predictors. Using the previously determined optimal training set size of 80%, the data is split into training, validation, and test sets. For each value of τ^2, Bayesian Linear Regression is performed with corresponding prior parameters, and the posterior mean of the coefficients is used to make predictions. The RMSE is calculated for both the training and validation sets to assess the model's fitting ability and generalization performance. Finally, the results are compiled into a data frame for comparison, providing insights into how the choice of prior variance affects the model's performance.

```r
# Load the necessary package "dplyr" for data manipulation functions.
library(dplyr)

#### Step 1: Prepare the Feature Matrix and Target Vector ####
# In this step, we set up the predictors (X) and response variable (y)
# for the entire dataset.

# Select all columns except "medv" as predictors and convert to matrix format.
X <- as.matrix(Boston %>% dplyr::select(-medv))  # "medv" is the target variable
# Set the response variable as the "medv" column.
y <- Boston$medv  # Target variable (median house values)
# Add an intercept term to the feature matrix.
X <- cbind(Intercept = 1, X)  # Adds a column of ones for the intercept term.

#### Step 2: Define Prior Variances to Test ####
# In this step, we specify a range of prior variances (tau_squared) to
# test in the model.
tau_squared_values <- c(0.01, 0.1, 1, 10, 100, 1000)  # Different values of tau_squared.

#### Step 3: Initialize Vectors to Store RMSE Values ####
# In this step, we prepare storage for RMSE values computed for each prior variance
rmse_train_hyper <- numeric(length(tau_squared_values))  # Store RMSE for training data
rmse_val_hyper <- numeric(length(tau_squared_values))    # Store RMSE for validation data

#### Step 4: Prepare Training, Validation, and Test Sets ####
# In this step, we split the data into training, validation,
# and test sets using the optimal training proportion.

# Set the optimal training set size (80%).
optimal_train_prop <- 0.8  # Proportion of the dataset used for training.

# Set a random seed for reproducibility.
set.seed(123)

# Total number of samples.
n_total <- nrow(X)
```

```r
# Randomly shuffle indices for splitting data.
indices <- sample(1:n_total)

# Calculate sizes of training, validation, and test sets.
n_train <- floor(optimal_train_prop * n_total)  # Number of training samples
n_remaining <- n_total - n_train                # Remaining samples for validation and test
n_val <- floor(n_remaining / 2)                 # Number of validation samples
n_test <- n_remaining - n_val                   # Number of test samples

# Split data into training, validation, and test sets.
train_indices <- indices[1:n_train]
val_indices <- indices[(n_train + 1):(n_train + n_val)]
test_indices <- indices[(n_train + n_val + 1):n_total]

# Create training set.
X_train <- X[train_indices, ]
y_train <- y[train_indices]

# Create validation set.
X_val <- X[val_indices, ]
y_val <- y[val_indices]

# Create test set.
X_test <- X[test_indices, ]
y_test <- y[test_indices]

#### Step 5: Loop Over Different Prior Variances (tau_squared) ####
# In this step, we evaluate model performance for each tau_squared value.
for (i in seq_along(tau_squared_values)) {
  tau_squared <- tau_squared_values[i]  # Current tau_squared value

  # Define prior distributions for model parameters.
  beta0 <- rep(0, ncol(X))          # Prior mean vector for regression coefficients
  V0 <- diag(tau_squared, ncol(X))  # Prior covariance matrix scaled by tau_squared
  a0 <- 0.01                  # Shape parameter for inverse-gamma prior on variance
  b0 <- 0.01                  # Scale parameter for inverse-gamma prior on variance

  # Set parameters for the Gibbs sampling algorithm.
  n_iter <- 5000 # Total number of iterations
  burn_in <- 1000 # Number of burn-in samples to discard

  # Fit the model and obtain posterior samples.
  results <- bayesian_linear_regression(
    X_train, y_train, beta0, V0, a0, b0,
    n_iter = n_iter, burn_in = burn_in
  )

  # Estimate regression coefficients using posterior means.
  beta_posterior_mean <- colMeans(results$beta_samples)  # Posterior mean estimates

  # Evaluate model performance on training data.
  # Predictions on training data
  y_pred_train <- X_train %*% beta_posterior_mean  # Predicted values for training data.
  rmse_train_hyper[i] <- sqrt(mean((y_train - y_pred_train)^2))  # RMSE for training data

  # Predictions on validation data
  y_pred_val <- X_val %*% beta_posterior_mean  # Predicted values for validation data
  rmse_val_hyper[i] <- sqrt(mean((y_val - y_pred_val)^2))  # Validation RMSE
}
#### Step 5: Combine RMSE Results into a Data Frame ####
# In this step , we organize RMSE results for further analysis and plotting.
rmse_hyper_results <- data.frame(
  TauSquared = tau_squared_values,      # TauSquared values tested
  RMSE_Train = rmse_train_hyper,        # RMSE values for training data
  RMSE_Validation = rmse_val_hyper      # RMSE values for validation data
)

# Print RMSE results
print(rmse_hyper_results)
```

```
##   TauSquared RMSE_Train RMSE_Validation
## 1      1e-02   7.005826        6.882651
## 2      1e-01   5.625432        5.281985
## 3      1e+00   4.958987        4.434398
## 4      1e+01   4.843736        4.307508
## 5      1e+02   4.679821        4.367329
## 6      1e+03   4.655981        4.444334
```

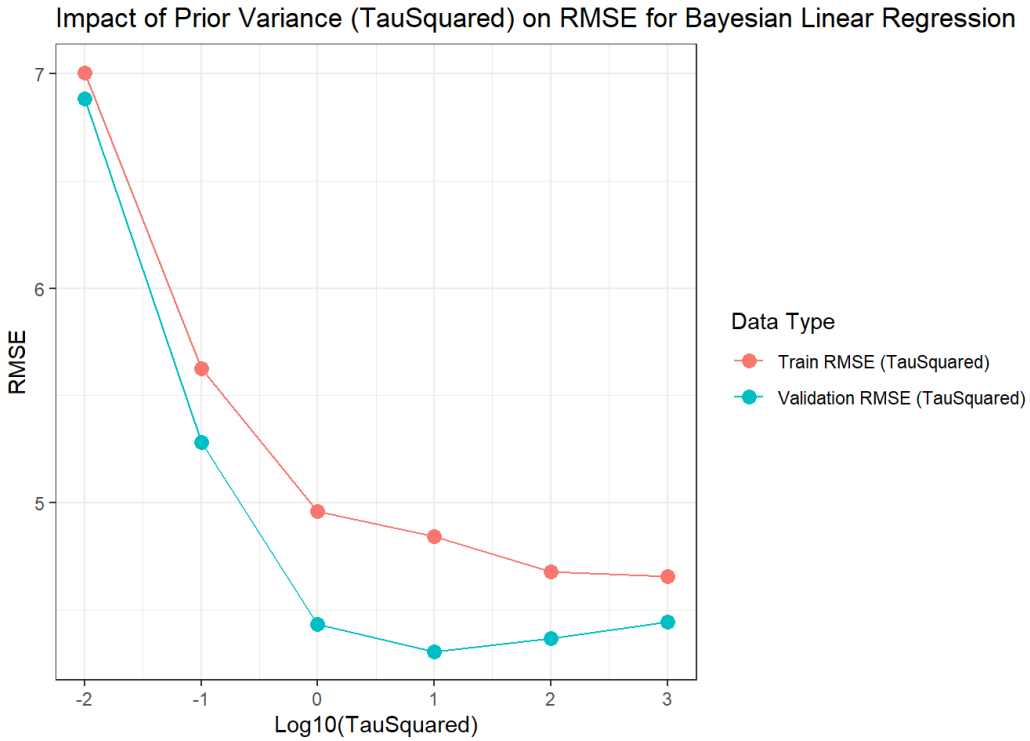
```
# Identify the optimal TauSquared value based on validation RMSE
optimal_index <- which.min(rmse_val_hyper)         # Index of the minimum validation RMSE
optimal_tau_squared <- tau_squared_values[optimal_index]  # Optimal TauSquared value
optimal_rmse <- rmse_val_hyper[optimal_index]      # Minimum validation RMSE

# Output the optimal TauSquared value and corresponding validation RMSE
cat("Optimal TauSquared Value:", optimal_tau_squared, "\n")
```

```
## Optimal TauSquared Value: 10
```

```
cat("Validation RMSE at Optimal TauSquared Value:", optimal_rmse, "\n")
```

```
## Validation RMSE at Optimal TauSquared Value: 4.307508
```

```
# Load necessary packages
library(reshape2)
library(dplyr)
library(ggplot2)

# Melt the RMSE results into a long format
rmse_hyper_melted <- melt(
  rmse_hyper_results,
  id.vars = "TauSquared",
  variable.name = "Dataset",
  value.name = "RMSE"
)

# Convert the wide-format data frame into a long-format data frame,

# Modify the Dataset column for descriptive labels
rmse_hyper_melted$Dataset <- recode(
  rmse_hyper_melted$Dataset,                  # The Dataset column in the melted data frame
  "RMSE_Train" = "Train RMSE (TauSquared)",
  "RMSE_Validation" = "Validation RMSE (TauSquared)"
)


# Plot RMSE vs. TauSquared for Training and Validation using ggplot2
tau_squared_rmse_plot <- ggplot(
  rmse_hyper_melted,
  aes(x = log10(TauSquared), y = RMSE, color = Dataset)
) +
  geom_point(size = 3) +
  geom_line() +
  labs(
    title = "Impact of Prior Variance (TauSquared) on RMSE for Bayesian Linear Regression",  # Plot title
    x = "Log10(TauSquared)",
    y = "RMSE",
    color = "Data Type"
  ) +
  theme_bw()


# Print the plot
print(tau_squared_rmse_plot)
```



section 5: Further Exploring Model Performance

The purpose of this section is to evaluate the performance of a Bayesian Linear Regression model on a test set using an optimal prior variance ($\tau^2 = 10$) determined from prior analysis. The process begins by merging the training and validation sets to ensure the model is trained on all available data except the test set, enhancing robustness. The prior parameters, including a zero-initialized mean vector for β and a diagonal covariance matrix scaled by the optimal τ^2, are defined alongside inverse-gamma hyperparameters for σ^2. Gibbs sampling is then performed to estimate the posterior distributions of β and σ^2, with a burn-in period to discard initial samples and ensure convergence. The posterior mean of β is extracted to make predictions on the test set, and the model's accuracy is quantified using the Root Mean Square Error (RMSE), which reflects the average deviation of predicted values from the true test values. The test RMSE is finally printed to summarize the model's predictive performance.

```
#### Step 1: Merge training and validation sets ####
# In this step, we create a combined dataset for model retraining.
# This ensures the model uses all available data (except the test set)
# for a more robust evaluation.

# Combine rows of training and validation predictors
X_train_full <- rbind(X_train, X_val)
# Combine responses from training and validation sets
y_train_full <- c(y_train, y_val)

#### Step 2: Set Prior Parameters with Optimal Tau Squared ####
# In this step, we define the prior distributions for the model parameters using # the optimal tau_squared value.

# Prior mean vector for regression coefficients (initialized to zeros).
beta0 <- rep(0, ncol(X))

# Prior covariance matrix for beta with optimal tau_squared = 10
V0 <- diag(10, ncol(X))

# Shape and scale parameters for the inverse-gamma prior on sigma^2.
a0 <- 0.01 # Shape parameter for inverse-gamma prior on sigma^2
b0 <- 0.01 # Scale parameter for inverse-gamma prior on sigma^2

#### Step 3: Set Gibbs Sampling Parameters ####
# In this step, we specify the number of iterations and burn-in period
# for the Gibbs sampler.

n_iter <- 5000 # Total number of iterations
burn_in <- 1000 # Number of burn-in iterations to discard

# Set a random seed for reproducibility
set.seed(123)

#### Step 4: Retrain the Model on the Combined Training and Validation Sets ####
# In this step, we fit the Bayesian Linear Regression model using
# the combined dataset.
```

```
results_final <- bayesian_linear_regression(
  X_train_full, y_train_full, beta0, V0, a0, b0,  # Input data and prior parameters
  n_iter = n_iter, burn_in = burn_in # Sampling parameters
)

#### Step 5: Obtain Posterior Mean of Beta ####
# We compute the posterior mean of the regression coefficients.
beta_posterior_mean_final <- colMeans(results_final$beta_samples)

#### Step 6: Predict on the Test Set ####
# We use the estimated coefficients to predict the response variable
# on the test set.
y_pred_test <- X_test %*% beta_posterior_mean_final

#### Step 7: Calculate RMSE on the Test Set ####
# In this setp, we compute the Root Mean Squared Error (RMSE) to
# evaluate model performance.
rmse_test_final <- sqrt(mean((y_test - y_pred_test)^2))
# Print the test RMSE with optimal tau_squared
cat("Test RMSE with optimal tau_squared (10):", rmse_test_final, "\n")
```

```
## Test RMSE with optimal tau_squared (10): 5.456036
```

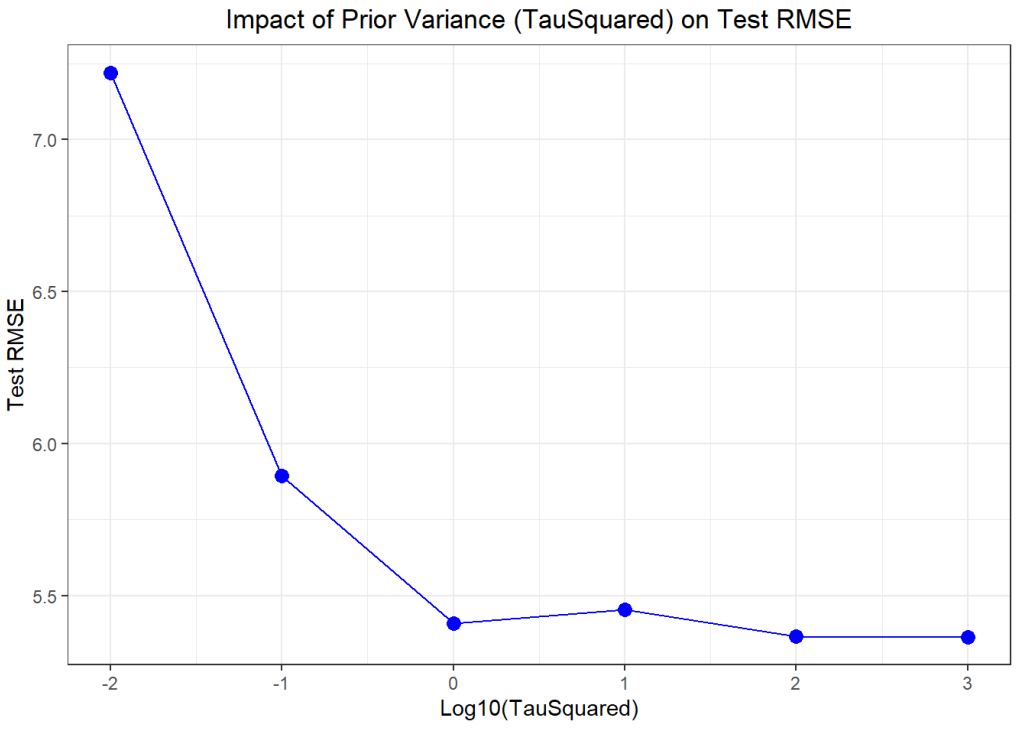
The purpose of this section is to evaluate the impact of different prior variances (τ^2) on the performance of a Bayesian Linear Regression model using the Boston dataset. The feature matrix (X) and target vector (y) are defined, with an intercept term added to X. Various τ^2 values are specified, and training and validation datasets are merged to maximize the data available for model training. For each τ^2, the model is trained using Gibbs sampling to estimate the posterior distributions of β and σ^2. The posterior mean of β is used to make predictions on the test set, and the Root Mean Square Error (RMSE) is calculated to measure prediction accuracy. Finally, create a data frame to summarize the test RMSE values for each τ^2, providing insights into how the choice of prior variance affects model performance.

```
# Load necessary package 'dplyr'
library(dplyr)

# The first step is define the feature matrix (X) and target vector (y) from the Boston dataset.
X <- as.matrix(Boston %>% dplyr::select(-medv))  # Select predictors and convert to matrix
y <- Boston$medv  # Target variable (median home value)

# Add intercept term to feature matrix
X <- cbind(Intercept = 1, X)

# The second step is define prior variances to test
# Create a vector of different tau_squared values (prior variances for beta) to evaluate.
tau_squared_values <- c(0.01, 0.1, 1, 10, 100, 1000)

# Initialize vectors to store RMSE values for test data
# Explanation: Prepare a numeric vector to store RMSE results for each tau_squared value.
rmse_test_hyper <- numeric(length(tau_squared_values))

# The third step is merge training and validation datasets into a single dataset for model retraining.
X_train_full <- rbind(X_train, X_val)  # Combine predictors from training and validation sets
y_train_full <- c(y_train, y_val)  # Combine target values from training and validation sets

# Loop over different prior variances
# Evaluate model performance for each tau_squared value.
for (i in seq_along(tau_squared_values)) {
  tau_squared <- tau_squared_values[i]  # Select the current tau_squared value

  # Set prior parameters
  # Define the prior parameters for Bayesian linear regression:
  beta0 <- rep(0, ncol(X))  # Prior mean vector for beta
  V0 <- diag(tau_squared, ncol(X))  # Prior covariance matrix for beta
  a0 <- 0.01  # Shape parameter for inverse-gamma prior on sigma^2
  b0 <- 0.01  # Scale parameter for inverse-gamma prior on sigma^2


  # Define the total number of iterations for Gibbs sampling and burn-in period.
  n_iter <- 5000  # Total number of iterations
  burn_in <- 1000  # Number of burn-in iterations to discard

  # Set random seed for reproducibility
  set.seed(123 + i)  # Set different seed for each iteration

 # Retrain the model on the combined training and validation sets
```

```
  results_final <- bayesian_linear_regression(
    X_train_full, y_train_full, beta0, V0, a0, b0,
    n_iter = n_iter, burn_in = burn_in
  )

  # Obtain Compute the posterior mean of beta from the Gibbs sampling results.
  # This mean is used for making predictions on the test data.
  beta_posterior_mean_final <- colMeans(results_final$beta_samples)

  # Predict the target variable on the test data using the posterior mean of beta.
  y_pred_test <- X_test %*% beta_posterior_mean_final

  # Compute the Root Mean Square Error (RMSE) to evaluate the model's prediction accuracy on the test set.
  rmse_test_hyper[i] <- sqrt(mean((y_test - y_pred_test)^2))  # RMSE formula
}

# Lastly, create a data frame to store tau_squared values and their corresponding test RMSE.
rmse_test_results <- data.frame(
  TauSquared = tau_squared_values,  # Prior variances (tau_squared values)
  RMSE_Test = rmse_test_hyper  # Test RMSE values
)

# Print the test RMSE values for each tau_squared value.
print(rmse_test_results)
```

```
##   TauSquared RMSE_Test
## 1      1e-02  7.219654
## 2      1e-01  5.893419
## 3      1e+00  5.410184
## 4      1e+01  5.454303
## 5      1e+02  5.366311
## 6      1e+03  5.365310
```

```
# Load the package 'ggplot2'
library(ggplot2)

# Plot the test RMSE vs. TauSquared
tau_squared_test_plot <- ggplot(rmse_test_results, aes(x = log10(TauSquared), y = RMSE_Test)) +
  geom_point(size = 3, color = "blue") +
  geom_line(color = "blue") +
  labs(
    title = "Impact of Prior Variance (TauSquared) on Test RMSE",
    x = "Log10(TauSquared)",
    y = "Test RMSE"
  ) +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5)
  )

print(tau_squared_test_plot)
```

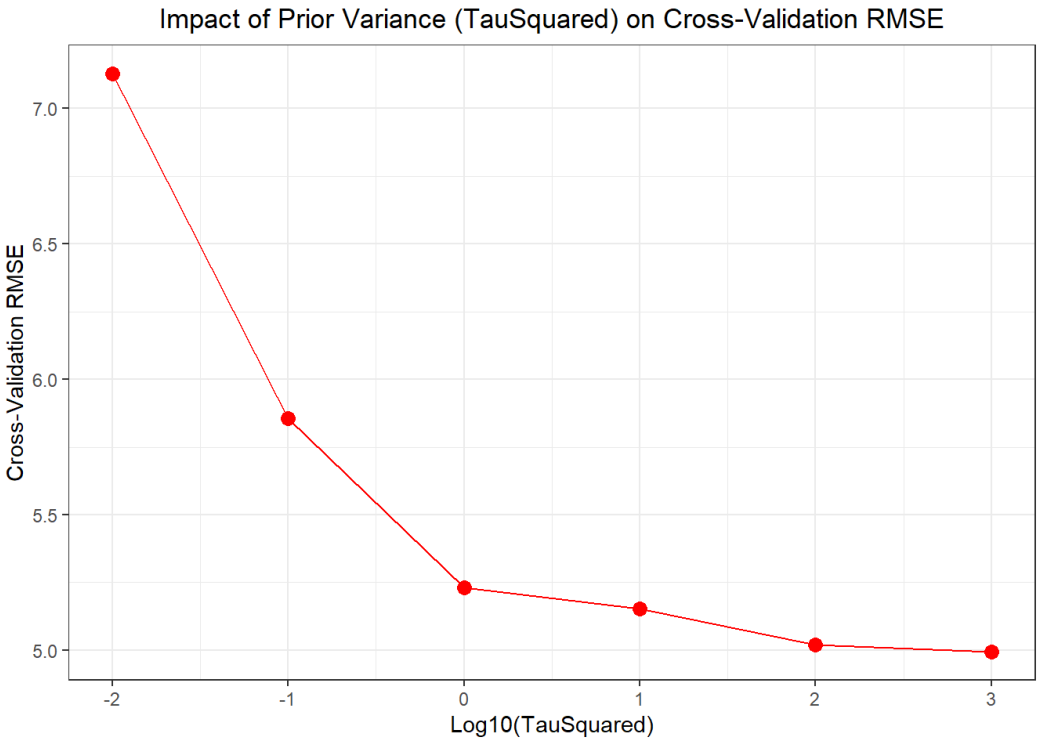


The purpose of this section is to evaluate the performance of different prior variances (τ^2) for Bayesian Linear Regression using k-fold cross-validation. The first setp is define the number of folds, and observations are randomly assigned to one of the k folds to ensure reproducibility. For each τ^2 value, the code iterates over the folds, splitting the data into training and validation sets, where one fold is used for validation and the remaining folds for training. The Bayesian model is trained on the training set using Gibbs sampling, and the posterior mean of β is used to predict the target variable on the validation set. The Root Mean Square Error (RMSE) is calculated for each fold to measure prediction accuracy. Finally, the mean RMSE across all folds is computed for each τ^2, providing a comprehensive evaluation of how prior variance affects model performance.

```r
#### Step 1: Set Up Cross-Validation Parameters ####
# In this step, we define the number of folds for k-fold cross-validation
k_folds <- 5  # Number of folds for cross-validation

# Set a random seed for reproducibility
set.seed(123)

#### Step 2: Create Fold Assignments ####
# In this step, we randomly assign each observation to one of the k folds.
n_total <- nrow(X)  # Total number of observations (ensure X is defined)
folds <- sample(rep(1:k_folds, length.out = n_total))  # Assign observations to folds

#### Step 3: Initialize Storage for RMSE Values ####
# In this step, we create a matrix to store RMSE values for
# each combination of tau_squared and fold.
tau_squared_values <- c(0.01, 0.1, 1, 10, 100, 1000)
rmse_cv <- matrix(0, nrow = length(tau_squared_values), ncol = k_folds)

# Notice that:
# - Rows correspond to different tau_squared values (hyperparameters).
# - Columns correspond to different folds in the cross-validation.

#### Step 4: Loop Over Tau Squared Values ####
# In this step, we evaluate model performance for each tau_squared value
# using cross-validation.
for (i in seq_along(tau_squared_values)) {
  tau_squared <- tau_squared_values[i]  # Select the current tau_squared value

  # Perform k-fold cross-validation for the current tau_squared value.
  for (k in 1:k_folds) {

    # Use fold 'k' as the validation set and the remaining folds
    # as the training set.
    test_indices_cv <- which(folds == k)    # Indices for the validation set
    train_indices_cv <- which(folds != k)   # Indices for the training set

    # Subset training and validation data
    X_train_cv <- X[train_indices_cv, ]     # Training predictors
    y_train_cv <- y[train_indices_cv]       # Training response variable

    X_val_cv <- X[test_indices_cv, ]        # Validation predictors
    y_val_cv <- y[test_indices_cv]          # Validation response variable

    # Define prior parameters for Bayesian Linear Regression.
    beta0 <- rep(0, ncol(X))                # Prior mean vector for beta
    V0 <- diag(tau_squared, ncol(X))        # Prior covariance matrix for beta
    a0 <- 0.01                # Shape parameter for inverse-gamma prior on sigma^2
    b0 <- 0.01                # Scale parameter for inverse-gamma prior on sigma^2

    # Notice that:
    # - The prior parameters reflect theinitial beliefs about the parameters.
    # - Varying tau_squared allows us to assess the impact of the prior variance.

    # Set parameters for the Gibbs sampling algorithm.
    n_iter <- 5000          # Total number of iterations (ensure n_iter is defined)
    burn_in <- 1000         # Number of burn-in samples (ensure burn_in is defined)

    # Notice that:
    # - These parameters control the convergence and accuracy of the Gibbs sampler
    # - Burn-in period allows the Markov chain to reach its stationary distribution

    # Fit the model using the training set for the current fold.
    results_cv <- bayesian_linear_regression(
      X_train_cv, y_train_cv, beta0, V0, a0, b0,
      n_iter = n_iter, burn_in = burn_in
    )
```

```
    # Notice that:
    # - The model is trained using Gibbs sampling to obtain posterior samples.
    # - Training is done on the training set specific to the current fold.

    # Obtain the posterior mean of the regression coefficients.
    beta_posterior_mean_cv <- colMeans(results_cv$beta_samples)

    # Notice that The posterior mean of beta serves as a point estimate for predictions,
    # it averages over the posterior distribution of beta.

    # Purpose: Use the posterior mean of beta to predict on the validation set.
    y_pred_cv <- X_val_cv %*% beta_posterior_mean_cv

    # Notice that: Predictions are made by applying the estimated coefficients to the validation data.

    # Evaluate the model's prediction accuracy on the validation set.
    rmse_cv[i, k] <- sqrt(mean((y_val_cv - y_pred_cv)^2))

    # Notice that:
    # - RMSE measures the average magnitude of prediction errors.
    # - Lower RMSE indicates better predictive performance on the validation set.

  }
}

#### Step 5: Compute Mean RMSE Across Folds ####
# In this step, we aggregate the RMSE across all folds for each tau_squared value.
rmse_cv_mean <- rowMeans(rmse_cv)  # Average RMSE across folds for each tau_squared value

# Notice that the averaging RMSE across folds provides a more stable estimate of
# model performance, it reduces variability due to different data splits.

#### Step 6: Compile Results into Data Frame ####
# In this step, Organize the results for further analysis.
data_cv <- data.frame(
  tau_squared = tau_squared_values,
  rmse = rmse_cv_mean
)
```

```
# Load the package 'ggplot2'
library(ggplot2)

# Plot Cross-Validation RMSE vs. TauSquared
tau_squared_cv_plot <- ggplot(data_cv, aes(x = log10(tau_squared), y = rmse)) +
  geom_point(size = 3, color = "red") +
  geom_line(color = "red") +
  labs(
    title = "Impact of Prior Variance (TauSquared) on Cross-Validation RMSE",
    x = "Log10(TauSquared)",
    y = "Cross-Validation RMSE"
  ) +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5)
  )

print(tau_squared_cv_plot)
```



Section 6: Exploring the Bias-Variance Trade-off

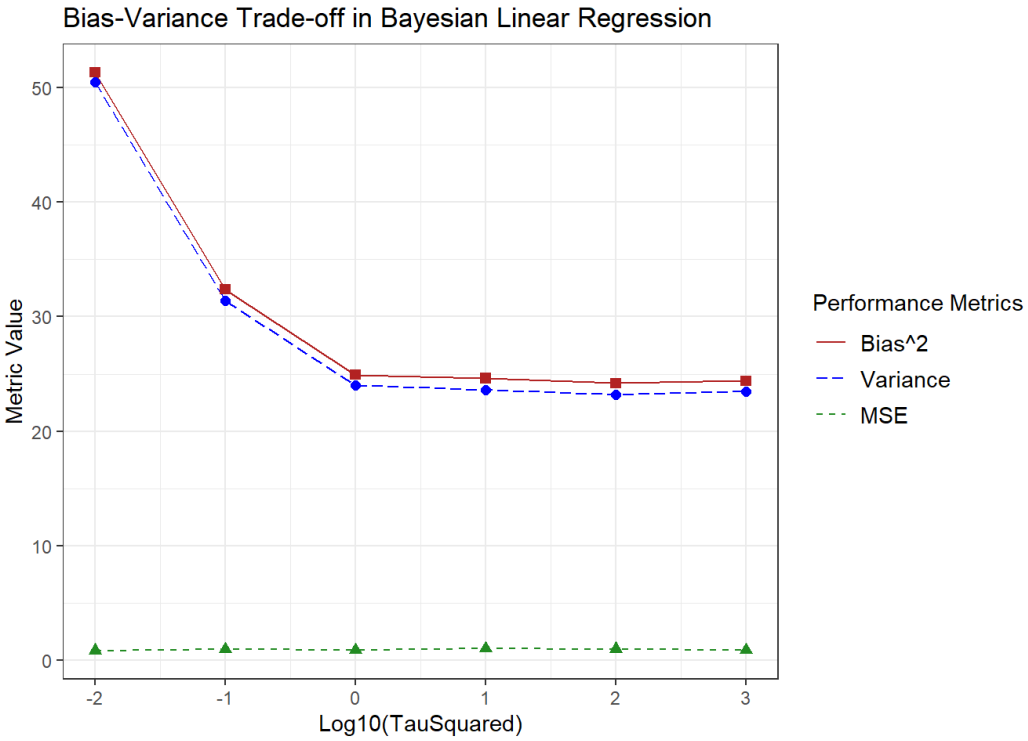
The purpose of this section is to evaluate the impact of prior variance (τ^2) on the bias-variance tradeoff and prediction accuracy in Bayesian Linear Regression. The dataset is prepared by defining the feature matrix and target vector, splitting it into training and validation sets, and specifying a range of τ^2 values to assess. For each τ^2, multiple simulations are performed using bootstrapped training samples, where a Bayesian model is fitted and predictions on the validation set are generated. The bias, variance, and mean squared error (MSE) are computed for each τ^2, quantifying the tradeoff between underfitting and overfitting. Results are consolidated into a data frame, enabling systematic comparison of model performance across different prior variances.

```r
# Load the necessary package "dplyr" for data manipulation functions.
library(dplyr)

#### Step 1: Prepare the Feature Matrix and Target Vector ####
# In this step , we set up the predictors ( X ) and response variable ( y )
# for the entire dataset .

# Select all columns except 'medv' as predictors and convert to matrix format.
X <- as.matrix(Boston %>% select(-medv))  # 'medv' is the target variable.
# Set the response variable as the 'medv' column.
y <- Boston$medv  # Target variable (median house values).
# Add an intercept term to the feature matrix.
X <- cbind(Intercept = 1, X)  # Adds a column of ones for the intercept term.

#### Step 2: Split Data into Training and Validation Sets ####
# In this step, we divide the dataset into training and validation sets based on the optimal training proportion.

# Set the optimal training set proportion 80%.
optimal_train_prop <- 0.8  # 80% of the data used for training.

# Set a random seed for reproducibility.
set.seed(123)

# Total number of observations.
n_total <- nrow(X)

# Randomly shuffle indices to randomize data splitting.
indices <- sample(1:n_total)

# Calculate the number of training and validation samples.
```

```r
n_train <- floor(optimal_train_prop * n_total)  # Number of training samples.
n_val <- n_total - n_train                      # Number of validation samples.

# Split indices for training and validation sets.
train_indices <- indices[1:n_train]             # Indices for training set.
val_indices <- indices[(n_train + 1):n_total]   # Indices for validation set.

# Create validation set (training set will be sampled during simulations).
X_val <- X[val_indices, ]                       # Validation predictors.
y_val <- y[val_indices]                         # Validation target values.

#### Step 4: Define Tau Squared Values ####
# In this step, we specify different prior variances (tau_squared)
# to evaluate their impact on the model.
tau_squared_values <- c(0.01, 0.1, 1, 10, 100, 1000)

# Initialize vectors to store bias, variance, and MSE for each tau_squared value.
bias_list <- numeric(length(tau_squared_values))
variance_list <- numeric(length(tau_squared_values))
mse_list <- numeric(length(tau_squared_values))

#### Step 5: Set Number of Simulations ####
# In this step, we fefine the number of simulations to perform for each tau_squared value.
n_simulations <- 50

#### Step 6: Evaluate Bias, Variance, and MSE for Each Tau Squared Value ####
# Loop over tau_squared values to compute bias, variance, and MSE.
for (i in seq_along(tau_squared_values)) {
  tau_squared <- tau_squared_values[i]  # Current tau_squared value.

  # Create a matrix to store predictions from each simulation.
  predictions_matrix <- matrix(0, nrow = nrow(X_val), ncol = n_simulations)

  # Estimate the model multiple times to compute bias and variance.
  for (sim in 1:n_simulations) {
    # Randomly sample training data with replacement (bootstrap sampling).
    train_sample_indices <- sample(train_indices, size = n_train, replace = TRUE)
    X_train <- X[train_sample_indices, ]  # Bootstrapped training predictors.
    y_train <- y[train_sample_indices]    # Bootstrapped training target values.

    # Purpose: Set prior distributions for Bayesian Linear Regression.
    beta0 <- rep(0, ncol(X))                # Prior mean vector for beta.
    V0 <- diag(tau_squared, ncol(X))        # Prior covariance matrix for beta.
    a0 <- 0.01                              # Shape parameter for inverse-gamma prior on sigma^2.
    b0 <- 0.01                              # Scale parameter for inverse-gamma prior on sigma^2.

    # Set parameters for the Gibbs sampling algorithm.
    n_iter <- 2000                          # Total number of iterations.
    burn_in <- 500                          # Number of burn-in iterations to discard.

    # Train the model using the bootstrapped training data.
    results <- bayesian_linear_regression(
      X_train, y_train, beta0, V0, a0, b0,
      n_iter = n_iter, burn_in = burn_in
    )

    # Obtain point estimates of the regression coefficients.
    beta_posterior_mean <- colMeans(results$beta_samples)

    # Use the estimated coefficients to predict on the validation set.
    y_pred_val <- X_val %*% beta_posterior_mean
    predictions_matrix[, sim] <- y_pred_val
  }

  # Calculate performance metrics based on the predictions:

  # Mean prediction for each validation sample across all simulations.
  y_pred_mean <- rowMeans(predictions_matrix)

  # Calculate the squared bias.
  bias_squared <- mean((y_pred_mean - y_val)^2)
  bias_list[i] <- bias_squared  # Store bias for the current tau_squared.
```

```r
  # Calculate the variance of the predictions.
  variance <- mean(apply(predictions_matrix, 1, var))
  variance_list[i] <- variance  # Store variance for the current tau_squared.

  # Calculate the Mean Squared Error (MSE).
  mse <- bias_squared + variance
  mse_list[i] <- mse  # Store MSE for the current tau_squared.
}

#### Step 7: Compile Results into Data Frame ####
# In this step, we organize the bias, variance, and MSE results
# for further analysis.
bias_variance_df <- data.frame(
  TauSquared = tau_squared_values,
  BiasSquared = bias_list,
  Variance = variance_list,
  MSE = mse_list
)
```

```r
# Load necessary packages
library(reshape2)
library(ggplot2)

# Reshape the data frame for plotting
# Convert the wide-format data frame (bias_variance_df) into a long-format data frame
# with 'TauSquared' as the identifier, and 'Component' and 'Value' as columns.
bias_variance_melted <- melt(bias_variance_df, id.vars = 'TauSquared', variable.name = 'Component', value.name =
'Value')

# Plot the bias_variance_plot
bias_variance_plot <- ggplot(bias_variance_melted, aes(x = log10(TauSquared), y = Value, color = Component, linet
ype = Component, shape = Component)) +
  geom_line() +
  geom_point(size = 2, show.legend = FALSE) +
  theme_bw() +
  ggtitle("Bias-Variance Trade-off in Bayesian Linear Regression") +
  xlab("Log10(TauSquared)") +
  ylab("Metric Value") +
  scale_color_manual(
    values = c("BiasSquared" = "blue",
               "Variance" = "forestgreen",
               "MSE" = "firebrick"),
    breaks = c("MSE", "BiasSquared", "Variance"),
    labels = c("Bias^2", "Variance", "MSE")
  ) +
  scale_linetype_manual(
    values = c("BiasSquared" = "longdash",
               "Variance" = "dashed",
               "MSE" = "solid"),
    breaks = c("MSE", "BiasSquared", "Variance"),
    labels = c("Bias^2", "Variance", "MSE")
  ) +
  scale_shape_manual(
    values = c("BiasSquared" = 16,
               "Variance" = 17,
               "MSE" = 15),
    labels = c("Bias^2", "Variance", "MSE"),
    breaks = c("MSE", "BiasSquared", "Variance"),
  ) +
  labs(
    color = "Performance Metrics",
    linetype = "Performance Metrics",
    shape = "Performance Metrics"
  ) +
  theme(
    legend.position = "right",
    legend.text = element_text(size = 11)
  )

print(bias_variance_plot)
```


Bias-Variance Trade-off in Bayesian Linear Regression
Metric Value
50
40
30
20
10
0
-2
-1
0
1
2
3
Log10(TauSquared)
Performance Metrics
Bias^2
Variance
MSE