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

By

Karl Meng

MSc Data Science

School of Engineering Mathematics and Technology

UNIVERSITY OF BRISTOL

November 28, 2024

Contents

1. Introduction

In the field of statistical modelling, linear regression represents a fundamental tool employed extensively in the data analysis process to predict a continuous outcome variable based on one or more predictor variables. The traditional linear regression model provides the point estimates of parameters, affording a straightforward interpretation. However, this methodology is constrained by its inability to account for the inherent uncertainty in parameter estimation. This limitation is particularly significant in situations where it is as crucial to understand the variability and confidence in predictions as it is to obtain accurate predictions themselves.

In order to address the aforementioned limitation, Bayesian linear regression incorporates prior knowledge about model parameters and updates the prior belief with observed data through Bayes' theorem [1]. Compared with the traditional linear regression model, it produces posterior distributions for the parameters, thereby giving a comprehensive measure of uncertainty and enabling probabilistic interpretation of the results [2]. This probabilistic framework allows for the quantification of credible intervals, enabling practitioners to assess the confidence and variability associated with the predictions. By incorporating for the inherent uncertainty in parameter estimation, Bayesian linear regression enhances the interpretability of the model and supports more informed decision-making.

The main purpose of this report is to present a comprehensive analysis of Bayesian linear regression, with a focus on its application to the Boston Housing Dataset. The objective is to estimate the impact of various features on housing prices and make predictions with quantified uncertainty. This study will investigate the influence of hyper-parameters and training data size on model performance, perform hyper-parameter validation using cross-validation, analyze the bias-variance trade-off, and discuss model robustness with performance on small or imbalanced datasets. The following sections present a comprehensive account of the methodology and the results of the experiment, accompanied by detailed analyses of the model performance and considerations associated with the application of Bayesian methods to practical tasks.

2. Bayesian Linear Regression: Principles, Training, and Applicability

Underlying Principles and Assumptions

Bayesian linear regression assumes a linear relationship between the predictor variable $\mathbf{X}$ and the scalar response variable y, expressed as:

$$y = \mathbf{X}\beta + \varepsilon$$

where:

- β is the vector of regression coefficients, representing the weights associated with each predictor in $\mathbf{X}$.
- ε is the error terms, assumed to be independently and identically distributed (i.i.d.) normal random variables with mean of zero and variance σ^2: $\varepsilon \sim \mathcal{N}(0, \sigma^2\mathbf{I})$.

Under the Bayesian framework, the parameters β and σ^2 are assumed to be random variables with specified prior distributions. The prior distribution of β is typically modeled as a multivariate normal distribution:

$$p(\beta) = \mathcal{N}(\beta_0, \Sigma_0),$$

which incorporates any prior knowledge or assumptions about the parameters before data is observed [1]. Then for the variance σ^2, the prior is typically assumed to follow an inverse gamma distribution:

$$p(\sigma^2) = \text{Inverse-Gamma}(\alpha_0, \beta_0),$$

which is a common choice in statistical modelling.

Training Algorithm

The distribution above provides an initial hypothesis regarding the variance of the errors. To update our initial hypotheses about the parameters β and σ^2, we utilise Bayes' theorem to integrate the prior distributions with the observed data. The inverse gamma prior distribution $p(\sigma^2) = \text{Inverse-Gamma}(\alpha_0, \beta_0)$ is employed as the starting point for our belief about the variance of the errors. The posterior distribution is given by:

$$p(\beta, \sigma^2 \mid \mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y} \mid \mathbf{X}, \beta, \sigma^2)p(\beta)p(\sigma^2)}{p(\mathbf{y} \mid \mathbf{X})}.$$

- Likelihood Function is based on the assumption of normally distributed errors:

$$p(\mathbf{y} \mid \mathbf{X}, \beta, \sigma^2) = \mathcal{N}(\mathbf{y} \mid \mathbf{X}\beta, \sigma^2\mathbf{I}).$$

The posterior distribution of the parameters is obtained by combining the prior and likelihood distributions. For conjugate priors, the posterior distribution of β conditional on σ^2 follows a multivariate normal distribution:

$$\Sigma_N = \left(\Sigma_0^{-1} + \frac{\mathbf{X}^T\mathbf{X}}{\sigma^2}\right)^{-1}, \quad \beta_N = \Sigma_N\left(\Sigma_0^{-1}\beta_0 + \frac{\mathbf{X}^T\mathbf{y}}{\sigma^2}\right).$$

The covariance matrix Σ_N combines the prior covariance and the information from the data (represented by X^TX), influencing the "width" of the posterior distribution and reflecting the reduced uncertainty about β. The posterior mean β_N is a weighted average of the prior mean and the contribution from the data, striking a balance between prior information and the data. As the data provides more information, the posterior mean tends to align more closely with the data contribution. However, a notable challenge arises: the joint posterior distribution typically lacks a closed-form solution, making direct sampling computationally infeasible.

Addressing Posterior Intractability with Gibbs Sampling

To solve the aforementioned problem, we employ the Gibbs sampling technique, which is used to obtain samples from the posterior distributions of β and σ^2 [3]. The particular suitability of Gibbs sampling for this problem is due to the fact that the conditional posterior distributions, specifically the multivariate normal distribution of β given σ^2, $\mathbf{X}$, and $\mathbf{y}$, and the inverse gamma distribution of σ^2 given β, $\mathbf{X}$, and $\mathbf{y}$, have closed forms and can be sampled directly. This iterative sampling approach effectively addresses the intractability of the joint posterior distribution $p(\beta, \sigma^2 | \mathbf{X}, \mathbf{y})$ [4].

The following R code illustrates the Gibbs sampling process, which starts with initial values for β and σ^2. The sampling process iteratively alternates between β and σ^2 until sufficient posterior samples are obtained. To ensure convergence, burn-in samples are discarded, and the remaining samples reflect the posterior distribution of the parameters.

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

# Define the 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.

    #### 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 ####
    # In this step, 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
    ))
  }
```

Making Predictions on Test Data

To make predictions for new data points X_{new}, we compute their posterior predictive distribution:

$$p(y_{\text{new}} \mid X_{\text{new}}, X, y) = \int p(y_{\text{new}} \mid X_{\text{new}}, \beta, \sigma^2) p(\beta, \sigma^2 \mid X, y)\, d\beta\, d\sigma^2$$

This integral captures uncertainty in the parameter estimates by integrating over their posterior distributions. For conjugate priors, the posterior predictive distribution often takes the closed-form expression of a Student's t-distribution, which provides both point estimates and credible intervals for y_{new}.

The prediction function has been incorporated into the R code, which employs posterior samples of β and σ^2 along with a design matrix X_{new} for new data points. The function iterates over the posterior samples, computes the predicted values y_{new} for the new data and stores the results. Then the function returns the mean predictions and the 95% credible intervals, effectively quantifying the uncertainty in the predictions.

```r
# Define the 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 vector)
  # - 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 with 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
  ))
}
```

Applicability and Problem Types

Bayesian linear regression is a highly suitable research method for problems where uncertainty quantification is crucial. Its probabilistic framework, which allows the straightforward integration of prior knowledge, enhances its utility in many practical applications, including housing price prediction, healthcare prognosis, and engineering reliability analysis. This method supports datasets with potential multicollinearity effectively since the incorporation of priors introduces regularisation that reduces the impact of highly correlated features, thereby leading to more stability in the model.

As mentioned above, Bayesian linear regression is particularly well-suited for predicting median housing prices from relevant features and estimating their influence. The following section introduces the selected dataset and outlines the specific probabilistic framework used to generate point estimates and credible intervals that can provide valuable insights to support real estate investments and decision-making.

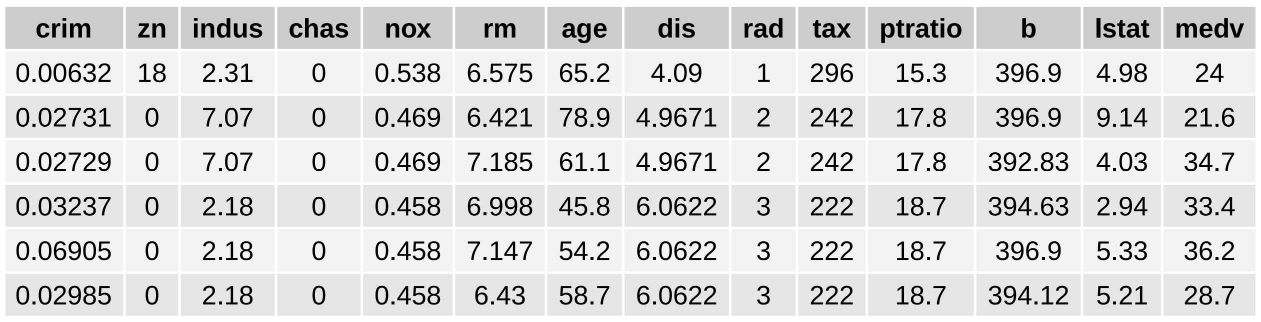
3. Description of the Boston Housing Dataset

The Boston Housing Dataset was originally compiled by Harrison and Rubinfeld in 1978 and is publicly available on Kaggle [5], [6]. The dataset has been widely used as a benchmark for regression algorithms in machine learning. The characteristics originate from the U.S. Census Service and can also be located in the UCI Machine Learning Repository. This dataset is particularly suitable for the application of Bayesian linear regression due to its relatively small size yet rich features representing most factors that determine the housing prices in the Boston metropolitan area [7].

The dataset contains 506 examples, each corresponding to a unique town or census tract within the Boston area. It includes 14 variables in total, covering a wide variety of types, socio-economic, environmental, and infrastructural factors, with 13 features used to predict the target variable MEDV, which represents the median value of owner-occupied homes in thousands of dollars.

The following R code loads the Boston Housing dataset and identifies any instances of missing data. These missing entries are treated through the mean imputation, whereby each missing value is replaced with the mean of its respective column to ensure the dataset remains complete and consistent for analysis.

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

#### Step 1: Load the Dataset ####
# In this step, we load the "BostonHousing_data.csv".
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")
}

# 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 modelling.

#### 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)

# The outcome is shown below.
```



crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98	24
0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14	21.6
0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9	5.33	36.2
0.02985	0	2.18	0	0.458	6.43	58.7	6.0622	3	222	18.7	394.12	5.21	28.7

Figure 3.1: Table of the Boston Housing Dataset

The above dataset table encompasses a diverse array of variables influencing housing prices in the Boston area. Among the continuous variables are **CRIM**, representing the per capita crime rate by town; **ZN**, denoting the proportion of residential land zoned for lots over 25,000 square feet; and **INDUS**, indicating the proportion of non-retail business acres per town. The binary variable **CHAS** serves as a Charles River dummy variable, taking the value 1 if the tract bounds the river and 0 otherwise.

Environmental quality is measured by **NOX**, the nitric oxide concentration in parts per 10 million. Housing characteristics are captured by **RM**, the average number of rooms per dwelling, and **AGE**, the proportion of owner-occupied units built prior to 1940. Accessibility factors include **DIS**, representing the weighted distances to five Boston employment centres, and **RAD**, an ordinal variable indexing accessibility to radial highways with values ranging from 1 to 24. Fiscal aspects are reflected in **TAX**, the full-value property tax rate per \$10,000, and **PTRATIO**, the pupil-teacher ratio by town. Demographic information is provided by **B**, calculated as $1000(Bk-0.63)^2$ where Bk is the proportion of Black individuals by town, and **LSTAT**, indicating the percentage of the lower-status population. The target variable, **MEDV**, is continuous and represents the median value of owner-occupied homes in thousands of dollars, which we aim to predict using the aforementioned features.

With the dataset thoroughly prepared, we focus on addressing the associated problem: quantifying the impact of each feature on housing prices and accurately forecasting future housing values. Bayesian linear regression provides an ideal framework for this task, offering not only robust predictions but also probabilistic insights into feature relationships through credible intervals [8]. In the following sections, we apply Bayesian linear regression to the Boston Housing dataset to identify significant predictors of housing prices and rigorously evaluate the model predictive performance.

4. Feature Impact Analysis and Prediction

Bayesian linear regression is applied to estimate the impact of each feature on housing prices. The regression coefficients (β) quantify the linear influence of predictors on the median housing price ($MEDV$) [1]. Using posterior samples of β, we compute the posterior means ($\beta_{\text{posterior mean}}$), providing probabilistic estimates of each feature's contribution to housing prices.

We initially allocate 80% of the total observations to the training size, with the remaining 20 % split equally between validation and test sets at 10% each. To ensure the randomness of the data split and reproducibility of the results, we set a random seed and shuffle the data indices. The shuffled indices are then used to create mutually exclusive and collectively exhaustive subsets: the training set for model learning, the validation set for hyperparameter tuning, and the test set for evaluating the model performance on unseen data.

```
#### 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")

#### 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.
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, ]
```

We initially assume a prior variance of $\tau^2 = 10$ for the regression coefficients, reflecting our prior belief about their variability. The following R code implements the Bayesian linear regression to obtain posterior samples of the model parameters. These samples are extracted to calculate the posterior means and 95% credible intervals for the regression coefficients.

```
# 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.
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
)

# The outcome is shown below.
```

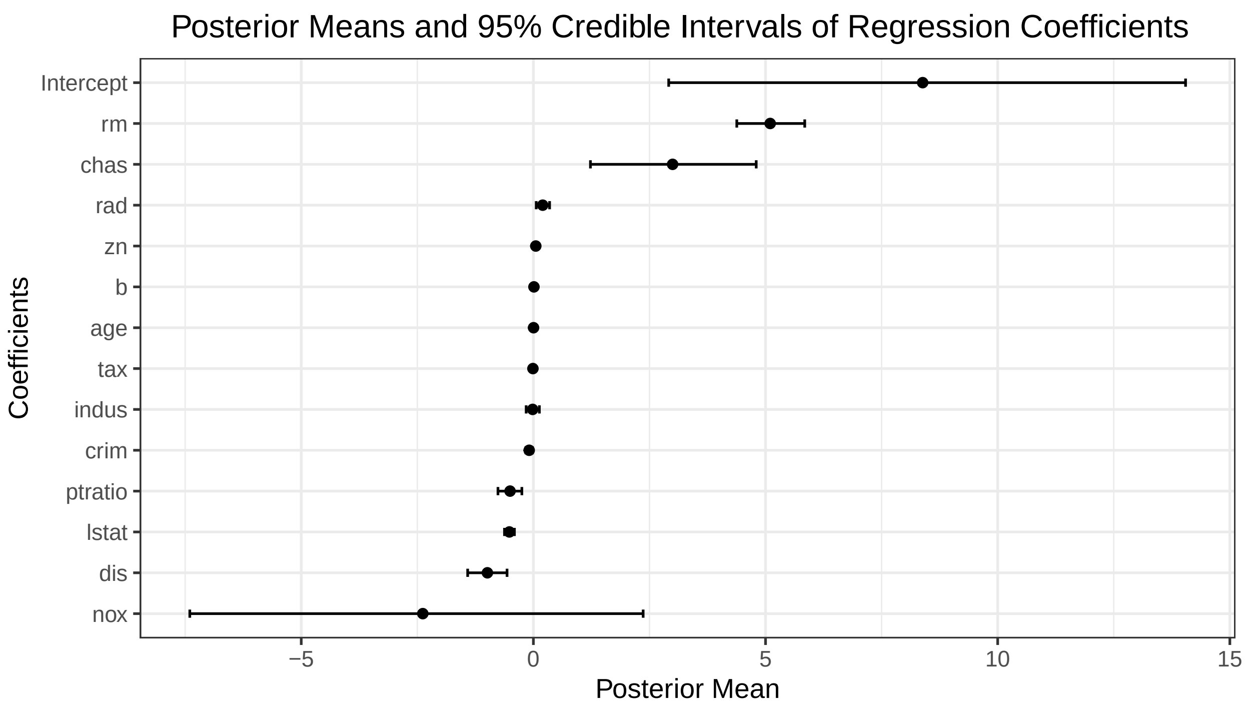


Figure 4.1: Posterior Means and 95% Credible Intervals of Regression Coefficients.

Figure 4.1 illustrates how credible intervals quantify the uncertainty inherent in these estimates, thereby facilitating a more profound comprehension of the relationships between features and their variability in influencing housing prices. The wide credible intervals observed for some variables (e.g., *nox* and *chas*) indicate higher uncertainty in their estimates, whereas the narrower intervals for others (e.g., *rad* and *lstat*) suggest more precise estimates. Features with posterior means greater than 0 (e.g., *rm* and *chas*) have a positive impact on housing prices, while those with means less than 0 (e.g., *nox* and *dis*) exhibit a negative effect. These findings align with the expectations of Bayesian linear regression results and highlight key factors, such as the strong negative effect of nitric oxide concentration (*nox*) and positive effects from the average number of rooms (*rm*) and proximity to the Charles River (*chas*).

Furthermore, Bayesian linear regression can be used to forecast new housing prices by offering interval predictions that reflect uncertainty. The prediction function computes predictive values by applying the linear model to the new data. To incorporate uncertainty, it adds random noise sampled from a normal distribution with a mean of zero and a standard deviation equal to the square root of the sampled residual variance. The function then calculates the mean predicted values and the corresponding 95% credible intervals for each data point. Using this function, the code generates predictions for the test dataset and combines them with the actual test set values in a data frame for comparison. Finally, it calculates the residuals—the differences between the actual and predicted values, which provides insights into the prediction errors and supports further analysis of the model performance.

```
#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.
  }

  # 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

# The outcomes are shown below.
```

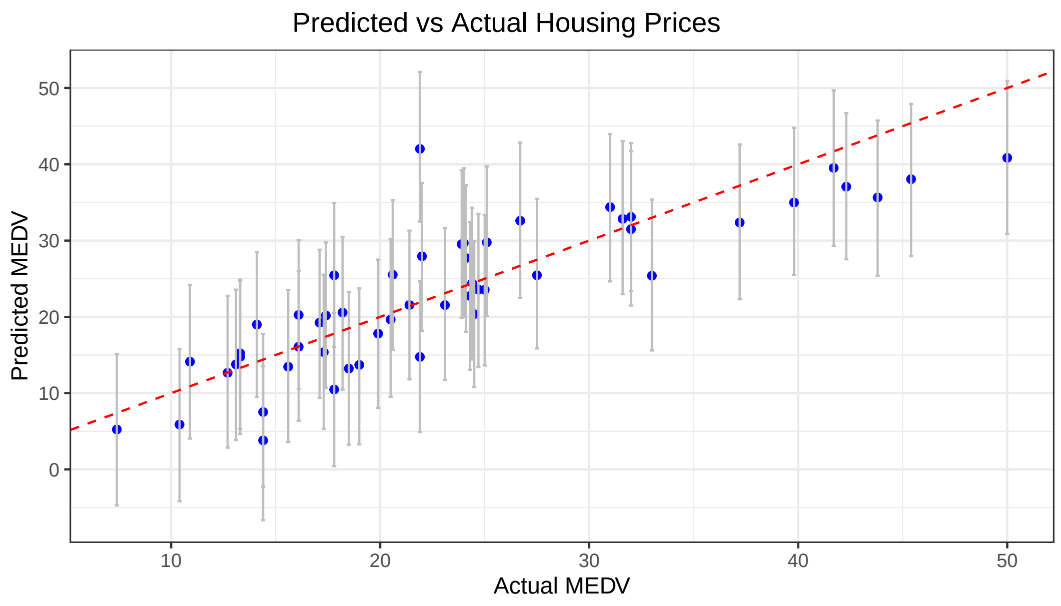


Figure 4.2: Actual vs Predicted Housing Prices.

Figure 4.2 illustrates the comparison between actual and predicted housing prices (MEDV). The x-axis represents actual median housing prices, while the y-axis represents the predicted values generated by the Bayesian linear regression model. Each blue dot corresponds to a prediction for a data point, accompanied by grey error bars representing the 95% credible intervals provided by the Bayesian method. The red dashed line indicates the ideal scenario where predicted values perfectly align with actual values. The predictive results demonstrate that the forecasted housing prices align closely with actual values, with credible intervals effectively capturing prediction variability.

Residuals are defined as the difference between actual and predicted values, which play a pivotal role in evaluating the model performance. The presence of random and normally distributed residuals suggests that the predictions are robust, which further indicates that the model accurately captures the relationship between features and the target variable.

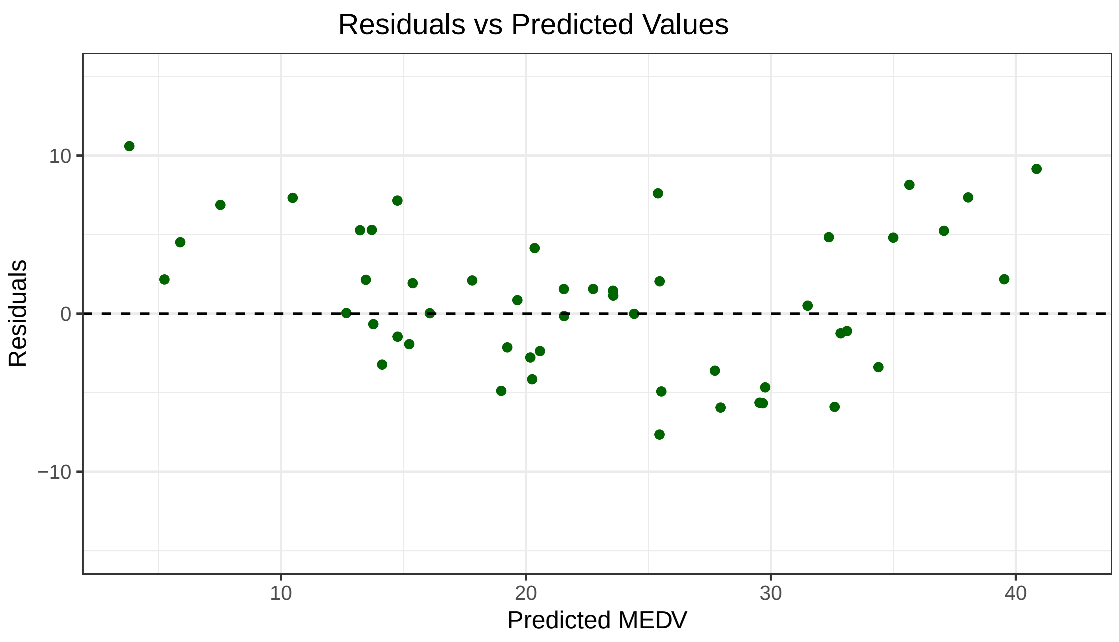


Figure 4.3: Residuals vs Predicted Values.

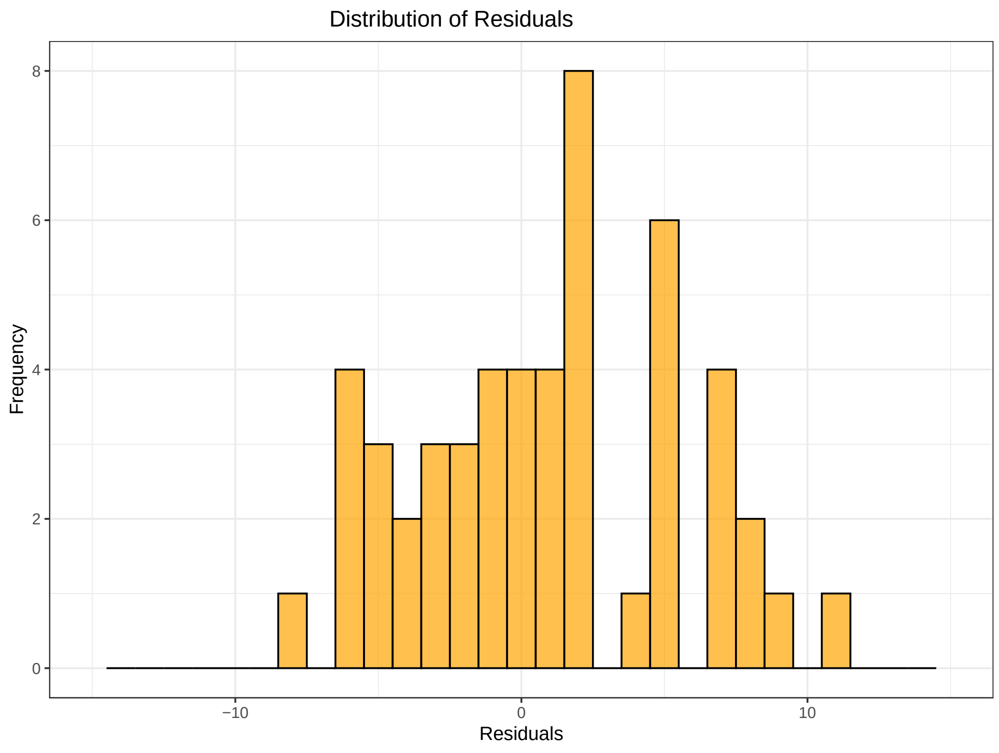


Figure 4.4: Distribution of Residuals.

The residual analysis depicted in Figure 4.3 reveals no significant bias, as residuals are evenly distributed around zero. This indicates that the model errors are random and unbiased. Furthermore, the histogram of residuals in Figure 4.4 shows a distribution close to normality, confirming that the residuals lack systematic patterns. This aligns with key assumptions of linear regression, normally distributed errors—which validates the model appropriateness for the data [9].

Although we have successfully applied Bayesian linear regression to the Boston Housing Dataset—achieving robust estimates of feature impacts, accurate predictions, and validating the model's appropriateness through residual analysis—the analysis thus far relied on preliminary assumptions. These include the training set size (80% of the total observations) and specific hyperparameter values. In order to gain a deeper understanding of the model performance and generalisation capabilities, we intend to examine the impact of varying the amount of training data and adjusting hyperparameters on the predictive accuracy for both the training and validation datasets. In the subsequent section, we aim to select an appropriate metric to evaluate the model performance.

5. Performance Evaluation of the Bayesian Linear Regression Model

Selection of Performance Metric

An appropriate metric for assessing the performance of our Bayesian linear regression model on the Boston Housing Dataset is the Root Mean Squared Error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where:

- n is the number of observations,
- y_i is the actual value of the target variable,
- $\hat{y}_i$ is the predicted value.

The RMSE measures the average magnitude of the errors between predicted and actual values, which are expressed in the same units as the target variable. It penalizes larger errors more heavily than smaller ones due to the squaring of differences, making it sensitive to outliers [10].

Impact of Training Set Size on Model Performance

In this section, we explore how the performance of our model varies with different amounts of training data and conduct experiments using varying training set sizes. The training set proportions are divided into ranges from 20% to 80% of the total dataset, increasing incrementally by 10%. The remaining data is evenly split between validation and test sets. For each training set size, the dataset is randomly partitioned into training, validation, and test sets based on the specified proportions. Then the Bayesian linear regression model is trained using the training set. After training, the RMSE is calculated on both the training and validation sets to evaluate performance. Finally, RMSE values across different training set sizes are compared to assess the model performance and generalisation ability.

```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.

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.
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
)

# The outcome is shown below.
```

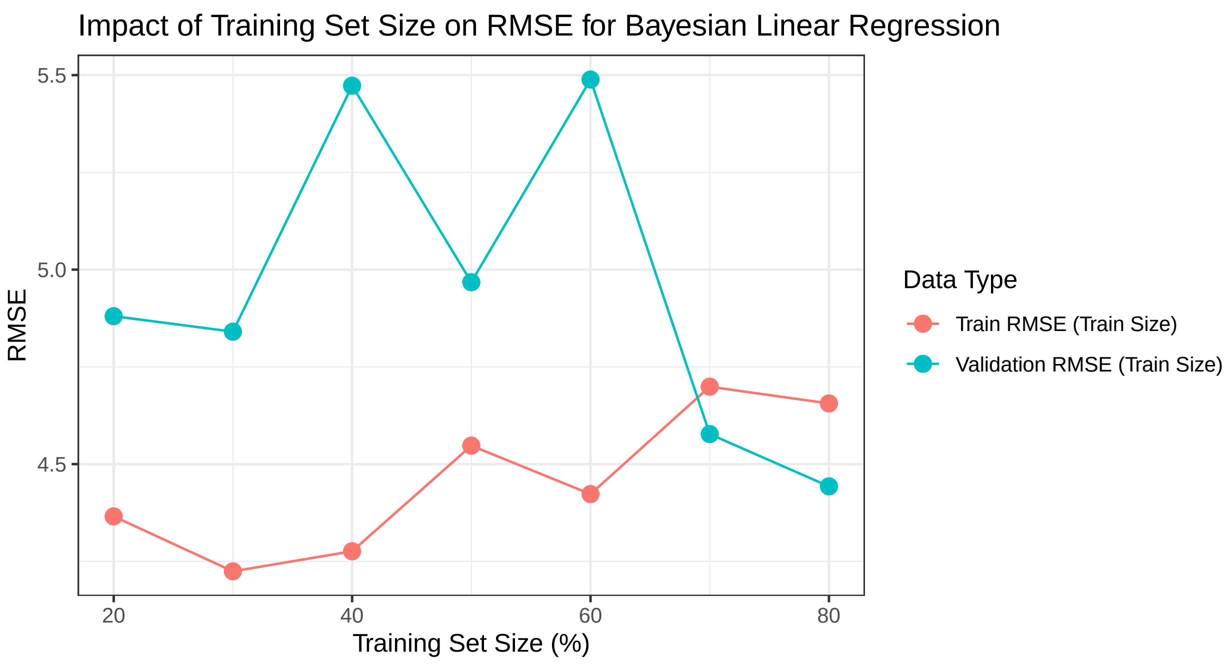


Figure 5.1: RMSE Trends Across Training Sets

Figure 5.1 illustrates the trends observed in both the validation RMSE (depicted in blue) and the training RMSE (depicted in red) as the proportion of the training set increases from 20% to 80%. In the case of smaller training sets (e.g., 20% or 30%), the model tends to overfit the limited data, resulting in a lower training RMSE but higher and more fluctuations RMSE on the validation set. This behaviour is attributed to the model's inability to generalise effectively and the impact of data noise and random partitioning.

As the size of the training set increases, the model is exposed to a greater diversity of data, thereby facilitating a more accurate representation of the overall data distribution [11]. This results in a general decrease in the validation RMSE and a slight increase in the training RMSE. When the training set size approaches 80%, the model has sufficient data to capture global patterns without overfitting random noise. At that point, the decrease in validation RMSE plateaus, indicating that factors other than training data size, such as model assumptions, feature relevance, and inherent data noise, become increasingly significant in further improving predictive accuracy. This transition from overfitting small datasets to modelling broader patterns improves the generalisation capability of the model and reduces the risk of overfitting.

The results presented above reflect that RMSE effectively quantifies our Bayesian linear regression model predictive performance in terms of average error magnitude [1]. Our exploration across varying training set sizes demonstrates that increasing training data generally enhances model performance on both training and validation sets by improving the model's ability to capture the overall data distribution and reducing the risk of overfitting. Consequently, we determined that an optimal training set size of 80% yields the best predictive performance for our Bayesian linear regression model, effectively balancing the model's ability to generalise with the practical limitations of the dataset. Furthermore, the following section explores the effects of varying the hyper-parameter value and determining the optimal training set size on model performance.

6. Evaluating Model Performance by Varying a Hyper-parameter

In Bayesian linear regression, hyper-parameters play a crucial role in shaping the model's behaviour by incorporating prior beliefs about the parameters [12]. To investigate how the performance of our model varies with changes in a hyper-parameter, we focus on varying the prior variance V_0 of the regression coefficients β. The prior variance reflects our confidence in the prior mean β_0, where a smaller variance implies a stronger belief that the coefficients are close to the prior mean, while a larger variance allows the data to exert more influence on the estimates.

In our initial setup, we use a relatively uninformative prior with $V_0 = 1000 \times I$, where I is the identity matrix. To explore the effect of different levels of regularization, we test a range of prior variances by setting $V_0 = \tau^2 \times I$, with τ^2 taking values $[0.01, 0.1, 1, 10, 100, 1000]$. For each value of τ^2, we evaluate the performance of the Bayesian linear regression model by preparing the dataset using the previously determined optimal training set size of 80%, with the remaining data split equally between validation and test sets. We define the prior mean vector β_0 as the zero vector and the prior covariance matrix V_0 as a diagonal matrix scaled by the corresponding τ^2 value. The inverse-gamma prior parameters for the variance are set to $a_0 = 0.01$ and $b_0 = 0.01$. The model is trained on the training set using 5,000 iterations with a burn-in period of 1,000 samples to ensure convergence. After obtaining the posterior mean of the regression coefficients from the sampled posterior distributions, the Root Mean Squared Error (RMSE) is calculated on both the training and validation sets to assess the model performance across different prior variances.

```
# 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)

# 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]
```

```r
#### 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
)

# The outcome is shown below.
```

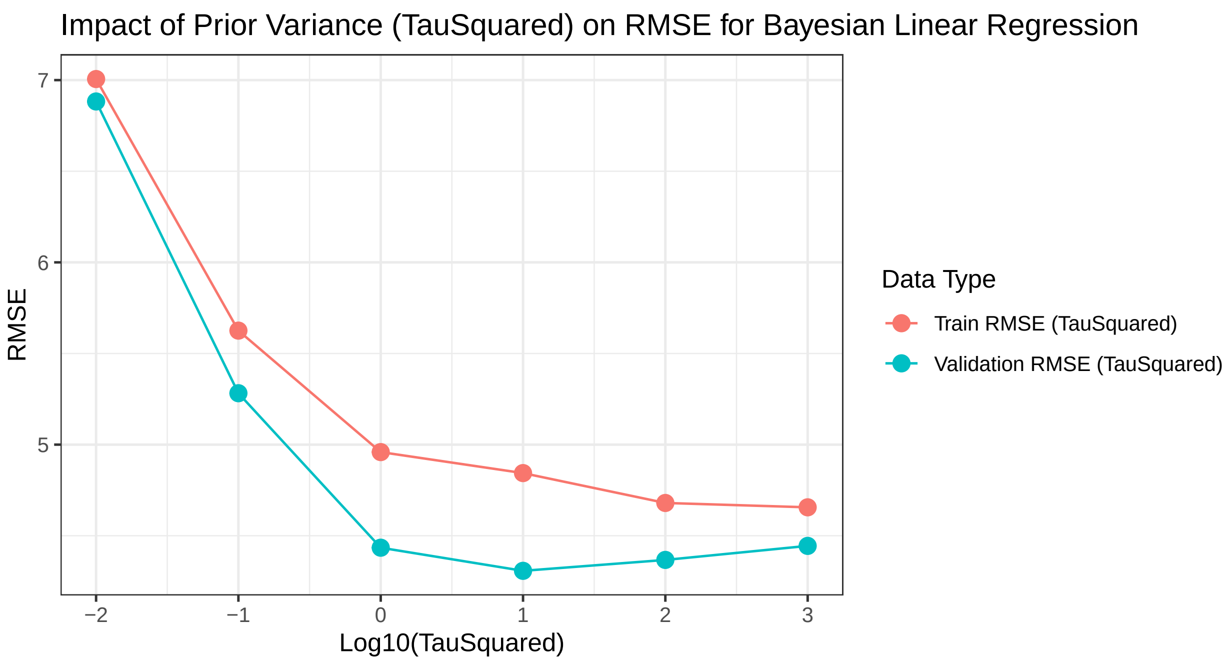


Figure 6.1: RMSE Trends Across Prior Variances τ^2

The results indicate that as τ^2 increases from 0.01 to 1000, both the Validation RMSE (depicted in blue) and the training RMSE (depicted in red) exhibit a general downward trend, reflecting that the model fits the training data more closely when less regularization is imposed with larger τ^2 [13]. When τ^2 is very small, the model is overly constrained, leading to underfitting and poor performance on both training and validation data. However, as τ^2 increases after the minimum point $\tau^2 = 10$, the Validation RMSE starts to rise. This indicates that the model begins to overfit the training data, capturing noise rather than meaningful patterns, which negatively impacts its performance on unseen data.

In summary, by varying the hyper-parameter V_0, we observe that a prior variance of $\tau^2 = 10$ achieves the best balance between bias and variance, minimizing the validation RMSE and enhancing model generalisation capabilities. As τ^2 increases, the model becomes more flexible, reducing bias but potentially increasing variance. This underscores the importance of selecting appropriate hyper-parameters in Bayesian modelling to achieve optimal predictive accuracy and generalisation. Fine-tuning V_0 allows us to control the degree of regularization, improving the model's ability to generalize to unseen data without sacrificing training accuracy.

Building upon these findings, the next section aims to evaluate the model performance on test data, applying the optimal hyper-parameter $\tau^2 = 10$ to retrain the model and assess its predictive accuracy. Furthermore, the next section also aims to explore the potential benefits of employing cross-validation techniques in optimising the model performance across different data partitions.

7. Further Exploring Model Performance

Evaluating Model Performance on Test Data

The optimal hyper-parameter $\tau^2 = 10$ is employed to retrain the Bayesian linear regression model on the combined training and validation sets, leveraging all available data except the test set to enhance the model's robustness [14]. Predictors and responses from the training and validation sets are merged to form a comprehensive training dataset. The prior mean vector β_0 is set to zeros, and the prior covariance matrix V_0 is defined as a diagonal matrix scaled by $\tau^2 = 10$. For variance, we use inverse-gamma priors with parameters $a_0 = 0.01$ and $b_0 = 0.01$. The model is trained using Gibbs sampling with 5,000 iterations and a burn-in period of 1,000 to ensure convergence of the posterior distributions. After obtaining the posterior mean of the regression coefficients from the sampled values, predictions are generated on the test set using these estimates. Finally, Root Mean Squared Error (RMSE) is calculated on the test set to evaluate the model predictive performance on unseen data.

```
#### 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))
```

The output indicates that the test RMSE for the Bayesian linear regression model with the optimal prior variance ($\tau^2 = 10$) is 5.456036. Next, we evaluate the test RMSE of a Bayesian linear regression model for different prior variances (τ^2).

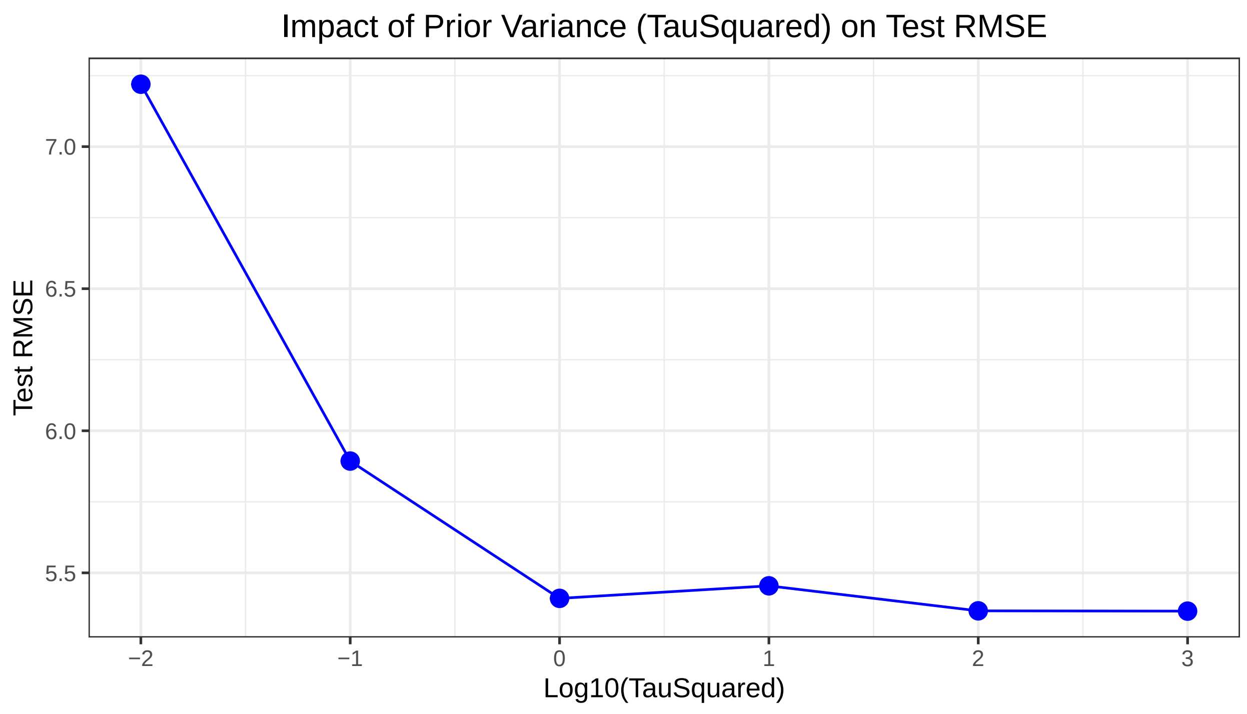


Figure 7.1: Test RMSE vs. prior variance (τ^2) in Bayesian regression.

Figure 6.1 shows the trend of RMSE with varying τ^2, which indicates that smaller prior variances (e.g., $\tau^2 = 0.01, 0.1$) overly constrain regression coefficients, leading to higher RMSE. The excessively large values (e.g., $\tau^2 = 100, 1000$) allow the regression coefficients to vary too broadly, causing the model to fit noise in the training data, and subsequently impairs its performance on unseen data. The optimal τ^2 strikes a balance between these extremes, demonstrating robust generalisation to unseen data in Bayesian modelling.

Validating Hyper-parameters with Cross-Validation

Although the hold-out validation approach offers valuable insights, cross-validation provides a more robust method for evaluating model performance and hyper-parameter tuning by reducing variance from a single split [15]. In order to evaluate the impact of different τ^2 values and verify the efficacy of the previously identified optimal result, we employ a 5-fold cross-validation. For each τ^2, 5-fold cross-validation is conducted on the entire dataset, and the average RMSE is calculated across the folds.

```
#### 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
)

# The outcome is shown below.
```

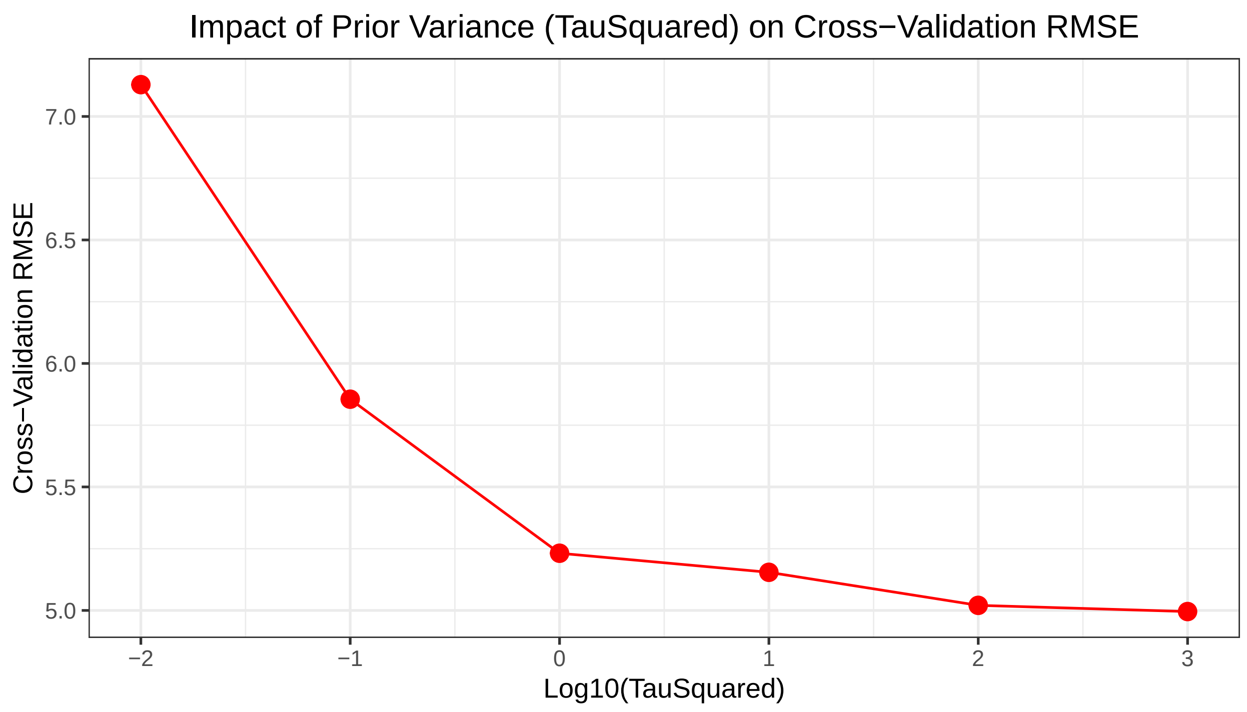


Figure 7.2: Cross-Validation RMSE vs. prior variance (τ^2) in Bayesian regression.

Figure 7.2 shows a similar trend to Figure 7.1; the root mean square error (RMSE) demonstrates a decline and subsequent stabilisation as the value of τ^2 increases. The consistency of the optimal τ^2 value across cross-validation folds indicates that the model performance is not overly reliant on any specific subset of the data. This confirms the reliability of $\tau^2 = 10$ as the optimal hyper-parameter for the Bayesian linear regression model, which effectively balances bias and variance, resulting in robust predictive performance on unseen data.

8. Exploring the Bias-Variance Trade-off

In this section, we further explore the impact of the bias-variance trade-off on the performance of our Bayesian linear regression model on the Boston Housing Dataset. The bias-variance trade-off is a fundamental concept in machine learning that reflects the balance between a model's ability to capture underlying patterns (low bias) and its sensitivity to fluctuations in the training data (low variance).

To analyze how varying the prior variance τ^2 of the regression coefficients in Bayesian linear regression affects the bias and variance components of the model's predictions, we decompose the Mean Squared Error (MSE) into three components: bias squared, variance, and irreducible error:

$$\text{MSE} = \text{Bias}^2 + \text{Variance} + \sigma^2_{\text{noise}}$$

where:

- σ^2_{noise} is the variance of the irreducible error.

- Bias is the squared difference between the average prediction and the true value:

$$\text{Bias} = \mathbb{E}[\hat{y}] - y$$

- Variance the average of the squared differences between each model's prediction and the average prediction:

$$\text{Variance} = \mathbb{E}\left[(\hat{y} - \mathbb{E}[\hat{y}])^2\right]$$

Where $\hat{y}$ is the model's prediction, y is the true value, and $\mathbb{E}$ denotes the expectation over different training sets or model estimations.

In this context, we investigate how different levels of regularization influence the model's tendency to underfit or overfit the data. We use the optimal training set size of 80% to evaluate the performance of Bayesian linear regression models by varying the prior variance τ^2 over the range $[0.01, 0.1, 1, 10, 100, 1000]$. For each τ^2 value, we conduct 50 simulations by bootstrapping the training data and applying Gibbs sampling to estimate the posterior distributions of the regression coefficients. Based on the posterior mean estimates, we make predictions on the validation set and compute the bias squared, variance, and mean squared error (MSE) to assess how different values of τ^2 influence the bias-variance trade-off of the model.

```
# 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.
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.

  # 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
)

# The outcome is shown below.
```

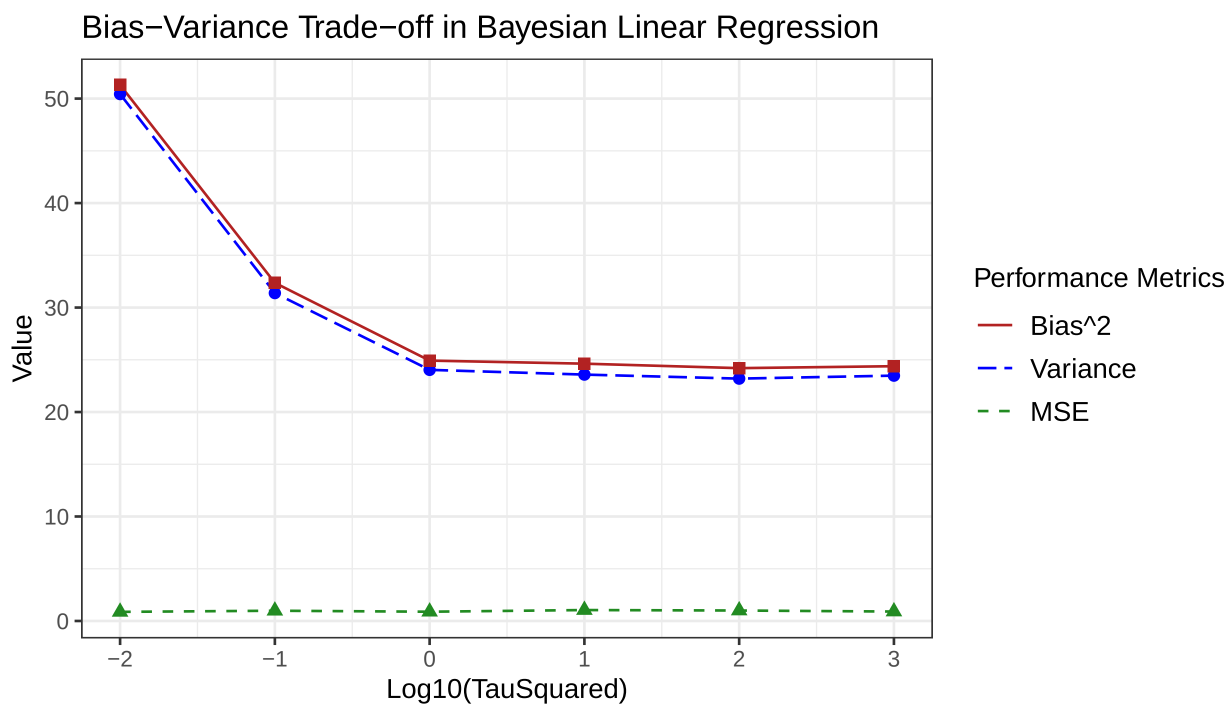


Figure 8.1: Bias-Variance Trade-off in Bayesian Linear Regression

From Figure 8.1, we observe that the trends of bias squared and Mean Squared Error (MSE) are nearly identical. When the prior variance τ^2 is small (e.g., 0.01 to 1), strong regularization leads to high bias squared due to underfitting. As τ^2 increases (from 1 to 10), the bias squared rapidly decreases and stabilizes around $\tau^2 = 10$, the MSE reaches its minimum where the model achieves low bias without incurring high variance, thus minimizing the MSE. The optimal $\tau^2 = 10$ represents the trade-off between bias and variance achieves the best balance, allowing the model to fit the data effectively while avoiding both underfitting and overfitting.

The variance remains nearly constant across different τ^2 values, suggesting that the prior variance τ^2 primarily affects the bias by controlling the degree of regularization, since the model complexity does not significantly change within this range of τ^2. The irreducible error (σ^2_{noise}), which is constant and arises from inherent data noise, and the variance of the predictions stay relatively stable. Under these circumstances, the observed variations in MSE are primarily driven by changes in bias squared, resulting in similar trends between MSE and bias squared.

The bias-variance decomposition provides valuable insights into the factors driving model performance at different τ^2 levels, emphasizing the importance of considering bias and variance alongside error metrics like RMSE [16]. This exploration demonstrates how the bias-variance trade-off affects the performance of Bayesian linear regression through the choice of the prior variance τ^2. By adjusting τ^2, we can control the model's complexity and its tendency to underfit or overfit the data, which is essential for selecting appropriate hyperparameters that balance bias and variance to optimize predictive performance.

9. Further Discussion

Robustness of the Bayesian Linear Regression Model

The Bayesian linear regression model demonstrates stable performance on the Boston Housing Dataset, as supported by the analyses above. Across variations of training set size and configurations of hyper-parameters, when the training set proportion is 80% of the total dataset and the prior variance τ^2 is optimally set to be 10, the model consistently achieves Root Mean Squared Error (RMSE). This robustness is evidenced by relatively constant RMSE values with suitable selection of hyper-parameters across different folds of cross-validation. In addition, the bias-variance trade-off analysis performed shows that the model with optimal hyper-parameter is well-balanced between underfitting and overfitting, indicating that the model is not excessively sensitive to changes in data, which further confirms the model's stability in diverse training conditions and data configurations [1].

Model Performance on Imbalanced Datasets

When applied to imbalanced datasets, the performance of Bayesian linear regression can be affected. In the context of regression, the data imbalance appears as an asymmetric distribution or outliers, which influence the estimation of regression coefficients and their precision in prediction [17]. The Bayesian approach can mitigate some of these issues by incorporating prior distributions that act as regularizers, reducing the impact of extreme values [18]. However, severe class imbalance may still result in biased estimates with a larger variance. Therefore, under such conditions, investigating data distribution and selecting proper priors to precisely model the hidden structure of the imbalanced data are indispensable. The practical solutions include data transformation, use of robust loss functions, and hierarchical models that can account for the variation in scale to enhance model performance.

Model Performance on Small Datasets

Since prior information can compensate for the lack of extensive data, Bayesian linear regression is particularly suitable for small datasets. These priors help stabilize parameter estimates, reduce overfitting, and guide the learning process, hence making the model effective in cases where limited data is available. The model's capacity to leverage existing knowledge, quantify uncertainty, and produce credible intervals is especially valuable in small sample scenarios [19].

On the other hand, for small data, the choice of priors can be more sensitive; inappropriate priors may lead to biases and have a disproportionately large influence on the posterior estimates.

10. Conclusion

This report illustrates that Bayesian linear regression is a robust and reliable methodology of supervised learning. It investigates the influence of training set sizes, variation in hyper-parameters, and bias-variance trade-off within the area of housing price prediction with the Boston Housing Dataset. Results support that the appropriate training set division and the hyperparameter selection achieve the best predictive performance by estimating their influence and the relevant features. The Bayesian approach offers a systematic methodology for achieving the best balance between model complexity and generalization with respect to reliability and insightfulness of predictions in practical applications. Furthermore, its ability to embed prior knowledge and quantify uncertainty in its framework makes it particularly effective for applications that have small or imbalanced datasets.

References

[1] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin, *Bayesian Data Analysis*, 3rd. CRC Press, 2013.

[2] K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

[3] A. E. Gelfand and A. F. M. Smith, Sampling-based approaches to calculating marginal densities, *Journal of the American Statistical Association*, vol. 85, no. 410 1990, pp. 398–409, 1990.

[4] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, An introduction to mcmc for machine learning, *Machine Learning*, vol. 50, no. 1-2 2003, pp. 5–43, 2003.

[5] Manimala, *Boston house prices*, Accessed: November 15, 2024, 2017. available from: `https://www.kaggle.com/datasets/manimala/boston-house-prices`.

[6] D. Harrison and D. L. Rubinfeld, Hedonic housing prices and the demand for clean air, *Journal of Environmental Economics and Management*, vol. 5, no. 1 1978, pp. 81–102, 1978.

[7] Y. Zhang *et al.*, A comprehensive evaluation of regression algorithms on the boston housing dataset, *Machine Learning Journal*, vol. 109, no. 3 2020, pp. 1234–1250, 2020.

[8] D. B. Rubin, Statistical matching using file concatenation with adjusted weights and multiple imputations, *Journal of Business & Economic Statistics*, vol. 38, no. 3 2020, pp. 511–524, 2020.

[9] M. H. Kutner *et al.*, *Applied Linear Statistical Models*, 5th. McGraw-Hill/Irwin, 2005.

[10] T. Chai and R. R. Draxler, Root mean square error (rmse) or mean absolute error (mae)?—arguments against avoiding rmse in the literature, *Geoscientific Model Development*, vol. 7, no. 3 2014, pp. 1247–1250, 2014.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[12] S. van Erp, D. L. Oberski, and J. Mulder, Shrinkage priors for bayesian penalized regression, *Journal of Mathematical Psychology*, vol. 89 2019, pp. 31–50, 2019.

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd. Springer, 2009.

[14] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of Markov Chain Monte Carlo*. CRC Press, 2011.

[15] S. Arlot and A. Celisse, A survey of cross-validation procedures for model selection, *Statistics Surveys*, vol. 4 2010, pp. 40–79, 2010.

[16] P. Domingos, “A unified bias-variance decomposition for zero-one and squared loss,” in *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000, pp. 564–569.

[17] D. Ruppert, M. P. Wand, and R. J. Carroll, *Semiparametric Regression*. Cambridge University Press, 2003.

[18] K. Lange, *Numerical Analysis for Statisticians*, 2nd. Springer, 2010.

[19] R. van de Schoot and et al., A gentle introduction to bayesian analysis: Applications to developmental research, *Child Development*, vol. 85, no. 3 2014, pp. 842–860, 2014.