Regression and Classification: A Comprehensive Analysis

Gabriele Durante

In this report, we aim to explore various statistical learning methodologies applied to regression and classification problems.

In the first part of the report, we examine linear regression models applied to aquatic toxicity data. This involves both traditional linear effects and alternative representations using dummy encoding, providing insight into how these transformations impact model fit and predictive accuracy. We apply techniques such as backward elimination and forward selection to perform variable selection, comparing criteria like AIC and BIC. Moreover, we introduce regularization techniques, specifically ridge regression, to address potential overfitting, followed by a discussion on parameter optimization via cross-validation and bootstrap methods. To extend our analysis to non-linear modeling, we apply generalized additive models (GAMs) with smoothing splines to capture more complex relationships within the data. Regression trees are explored, where we employ cost-complexity pruning to optimize tree size, emphasizing the balance between model complexity and prediction accuracy.

In the second part, we shift to a classification problem using the Pima Indians Diabetes dataset. Here, we experiment with k-nearest neighbors (k-NN), generalized additive models, classification trees, bagged trees, random forests, and neural networks. Each method is rigorously evaluated, using cross-validation techniques to assess generalization error and performance trade-offs. The report concludes with a comparative analysis of these models, offering insights into the advantages and limitations of different statistical learning approaches within the context of high-dimensional data.

Regression Model Comparison for Predicting Aquatic Toxicity

In this part, we investigate the acute aquatic toxicity of various organic molecules, as quantified by the lethal concentration (LC50) that induces mortality in 50% of the planktonic crustacean Daphnia magna over a 48-hour exposure period. The objective is to develop predictive models that leverage molecular descriptors to ascertain the toxicity levels of these compounds. To do this, we use a dataset comprising 546 observations, which is sourced from the UCI Machine Learning Repository. The dataset encompasses eight key molecular descriptors identified as significant predictors of LC50:

- TPSA: Topological Polar Surface Area, calculated via a contribution method considering the presence of nitrogen, oxygen, potassium, and sulfur.
- SAacc: Van der Waals Surface Area (VSA) of hydrogen bond acceptor atoms.
- H050: The count of hydrogen atoms bonded to heteroatoms.
- MLOGP: A measure of lipophilicity, serving as a critical determinant of narcosis.
- RDCHI: A topological index encapsulating information related to molecular size and branching.
- GATS1p: An indicator of molecular polarizability.
- nN: The total number of nitrogen atoms within the molecule.
- C040: The count of specific carbon atom types, including those found in esters, carboxylic acids, thioesters, carbamic acids, and nitriles.

```
summary(data)
```

```
##       TPSA            SAacc             H050             MLOGP
##  Min.   :  0.00   Min.   :  0.00   Min.   : 0.0000   Min.   :-6.446
##  1st Qu.: 15.79   1st Qu.: 11.00   1st Qu.: 0.0000   1st Qu.: 1.232
```

```
##  Median : 40.46   Median : 42.68   Median : 0.0000   Median : 2.273
##  Mean   : 48.47   Mean   : 58.87   Mean   : 0.9377   Mean   : 2.313
##  3rd Qu.: 70.02   3rd Qu.: 77.49   3rd Qu.: 1.0000   3rd Qu.: 3.393
##  Max.   :347.32   Max.   :571.95   Max.   :18.0000   Max.   : 9.148
##      RDCHI           GATS1p            nN              C040
##  Min.   :1.000   Min.   :0.281   Min.   : 0.000   Min.   : 0.0000
##  1st Qu.:1.975   1st Qu.:0.737   1st Qu.: 0.000   1st Qu.: 0.0000
##  Median :2.344   Median :1.020   Median : 1.000   Median : 0.0000
##  Mean   :2.492   Mean   :1.046   Mean   : 1.004   Mean   : 0.3535
##  3rd Qu.:2.911   3rd Qu.:1.266   3rd Qu.: 2.000   3rd Qu.: 0.0000
##  Max.   :6.439   Max.   :2.500   Max.   :11.000   Max.   :11.0000
##       LC50
##  Min.   : 0.122
##  1st Qu.: 3.602
##  Median : 4.516
##  Mean   : 4.658
##  3rd Qu.: 5.607
##  Max.   :10.047
```

```
colSums(is.na(data))
```

```
##   TPSA  SAacc   H050  MLOGP  RDCHI GATS1p     nN   C040   LC50
##      0      0      0      0      0      0      0      0      0
```

With a complete dataset comprising 546 observations, we can ensure that our modeling efforts are not hindered by data gaps. This completeness facilitates a more accurate assessment of the relationships between the molecular descriptors and LC50, allowing for a more confident interpretation of the findings and their implications in predicting aquatic toxicity.

Linear and Dummy Encoding Approach

To evaluate the predictive models effectively, we partition the dataset into training and testing subsets. A random seed is set using `set.seed(2024)` to ensure reproducibility. The training set is constructed by sampling approximately two-thirds of the total observations with the command `index <- sample(1:nrow(data), size = round(2/3 * nrow(data)))`.

- Training Data: `trainData <- data[index, ]` comprises the selected observations for model fitting.
- Testing Data: `testData <- data[-index, ]` contains the remaining observations for model evaluation.

```
# build the dataset for the training
set.seed(2024)
index <- sample(1:nrow(data), size = round(2/3 * nrow(data)))
trainData <- data[index, ]
testData <- data[-index, ]
```

Linear Model

First we fitted a linear regression model to predict LC50 using molecular descriptors from the training dataset. The model's performance is summarized as follows:

```
set.seed(2024)
### Modeling Count Variables Directly as Linear Effects
model1 <- lm(LC50 ~., data = trainData)

pred_model1_train <- predict(model1, newdata = trainData)
pred_model1_test <- predict(model1, newdata = testData)
```

```r
error_train <- mean((pred_model1_train - trainData$LC50)^2)
error_test <- mean((pred_model1_test - testData$LC50)^2)
```

```r
cat(paste(
  "Training Error (Linear):","\t", error_train, "\n",
  "Test Error (Linear):","\t","\t", error_test, "\n"))
```

```
## Training Error (Linear):      1.37716232228835
##  Test Error (Linear):         1.55697805869271
```

```r
summary(model1)
```

```
##
## Call:
## lm(formula = LC50 ~ ., data = trainData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.1890 -0.7482 -0.1153  0.6079  3.7987
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.875587   0.298304   9.640  < 2e-16 ***
## TPSA         0.026589   0.003220   8.258 2.96e-15 ***
## SAacc       -0.012127   0.002522  -4.809 2.25e-06 ***
## H050         0.031343   0.070945   0.442  0.65891
## MLOGP        0.496920   0.077134   6.442 3.83e-10 ***
## RDCHI        0.308016   0.166170   1.854  0.06462 .
## GATS1p      -0.537871   0.187291  -2.872  0.00433 **
## nN          -0.198318   0.057041  -3.477  0.00057 ***
## C040        -0.055902   0.090200  -0.620  0.53582
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.188 on 355 degrees of freedom
## Multiple R-squared:  0.4721, Adjusted R-squared:  0.4602
## F-statistic: 39.68 on 8 and 355 DF,  p-value: < 2.2e-16
```

The R-squared value of 0.472 indicates that while the linear model captures some of the variability in the data, it leaves a considerable amount unexplained. Highly significant predictors, such as TPSA ($p < 2e\text{-}16$) and MLOGP ($p < 3.83e\text{-}10$), reveal strong relationships with LC50, indicating that increased topological polar surface area and greater lipophilicity correlate with higher toxicity levels.

Dummy Linear Model

After, model dummy encoding was applied to the count variables nN, C040, and H050 in both the training and test datasets to transform these continuous variables into binary indicators. Specifically, values greater than zero were recoded to 1, while values of zero were recoded to 0. This approach allows the linear model to assess the presence or absence of these variables as factors influencing acute aquatic toxicity. Predictions were generated for both the training and test datasets, enabling the calculation of the training error and test error.

```r
### Dummy Encoding for Count Variables
train_dummy <- trainData
test_dummy <- testData

# dummy encoding
```

```r
train_dummy$nN <- ifelse(train_dummy$nN > 0, 1, 0)
test_dummy$nN <- ifelse(test_dummy$nN > 0, 1, 0)
test_dummy$C040 <- ifelse(test_dummy$C040 > 0, 1, 0)
train_dummy$C040 <- ifelse(train_dummy$C040 > 0, 1, 0)
test_dummy$H050 <- ifelse(test_dummy$H050 > 0, 1, 0)
train_dummy$H050 <- ifelse(train_dummy$H050 > 0, 1, 0)

model2 <- lm(train_dummy$LC50 ~ ., data = train_dummy)

pred_model2_train <- predict(model2, newdata = train_dummy)
pred_model2_test <- predict(model2, newdata = test_dummy)

error_train_dummy <- mean((pred_model2_train - train_dummy$LC50)^2)
error_test_dummy <- mean((pred_model2_test - test_dummy$LC50)^2)
```

```r
cat(paste(
  "Training Error (Dummy):","\t", error_train_dummy, "\n",
  "Test Error (Dummy):","\t\t", error_test_dummy, "\n"))
```

```
## Training Error (Dummy):   1.42423630476475
##  Test Error (Dummy):          1.61682541849219
```

```r
summary(model2)
```

```
##
## Call:
## lm(formula = train_dummy$LC50 ~ ., data = train_dummy)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.0143 -0.7807 -0.1313  0.6313  3.7856
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.007963   0.305982   9.831  < 2e-16 ***
## TPSA         0.022380   0.003185   7.026 1.09e-11 ***
## SAacc       -0.010184   0.002193  -4.643 4.83e-06 ***
## H050        -0.097746   0.154467  -0.633  0.52727
## MLOGP        0.502377   0.076886   6.534 2.22e-10 ***
## RDCHI        0.262639   0.167774   1.565  0.11837
## GATS1p      -0.559217   0.185302  -3.018  0.00273 **
## nN          -0.054983   0.149020  -0.369  0.71237
## C040        -0.162055   0.162002  -1.000  0.31783
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.208 on 355 degrees of freedom
## Multiple R-squared:  0.454,  Adjusted R-squared:  0.4417
## F-statistic:  36.9 on 8 and 355 DF,  p-value: < 2.2e-16
```

The training error for the dummy-encoded model is 1.424, while the test error is 1.617. Compared to the previous linear model, which yielded a training error of 1.377 and a test error of 1.557, this model exhibits slightly higher error rates on both the training and test datasets. This increase suggests that the dummy encoding may not have improved the model's predictive performance. This could indicate a potential loss of information due to the transformation of continuous variables into binary indicators, which could limit the

model's ability to capture the nuances of the underlying relationships between predictors and the response variable.

The overall model performance is reflected in the Multiple R-squared value of 0.454, while the adjusted R-squared of 0.4417 accounts for the number of predictors in the model. The F-statistic (36.9) with a p-value $<$ 2.2e-16 indicates that the model is statistically significant.

Empirical Error Distribution Analysis in Regression Models

The function `perform_analysis` is defined to conduct repeated evaluations of the two linear models defined before on the same dataset and returns the values of error test and train. The analysis is repeated 200 times using the `replicate` function, enabling the computation of average test errors for both models. The results are then visualized using a density plot, illustrating the empirical distribution of test errors for each model.

```
library(ggplot2)

perform_analysis <- function(data) {
  index <- sample(1:nrow(data), size = round(2/3 * nrow(data)))
  trainData <- data[index, ]
  testData <- data[-index, ]

  # model 1
  model1 <- lm(LC50 ~., data = trainData)
  pred_model1_train_200 <- predict(model1, newdata = trainData)
  pred_model1_test_200 <- predict(model1, newdata = testData)
  error_train_200_lm <- mean((pred_model1_train_200 - trainData$LC50)^2)
  error_test_200_lm <- mean((pred_model1_test_200 - testData$LC50)^2)

  # model 2
  train_dummy <- trainData
  test_dummy <- testData

  # dummy encoding
  train_dummy$nN <- ifelse(train_dummy$nN > 0, 1, 0)
  test_dummy$nN <- ifelse(test_dummy$nN > 0, 1, 0)
  test_dummy$C040 <- ifelse(test_dummy$C040 > 0, 1, 0)
  train_dummy$C040 <- ifelse(train_dummy$C040 > 0, 1, 0)
  test_dummy$H050 <- ifelse(test_dummy$H050 > 0, 1, 0)
  train_dummy$H050 <- ifelse(train_dummy$H050 > 0, 1, 0)

  model2 <- lm(train_dummy$LC50 ~ ., data = train_dummy)

  pred_model2_train <- predict(model2, newdata = train_dummy)
  pred_model2_test <- predict(model2, newdata = test_dummy)

  error_train_200_dummy <- mean((pred_model2_train - train_dummy$LC50)^2)
  error_test_200_dummy <- mean((pred_model2_test - test_dummy$LC50)^2)

  return(c(error_test_200_lm, error_test_200_dummy))
}
```

```
num_repeats <- 200
results <- replicate(num_repeats, perform_analysis(data))

avg_errors <- colMeans(results)
```

```r
results_df <- data.frame(
  Error = c(results[1, ], results[2, ]),
  Model = rep(c("Linear Effects", "Dummy Encoding"), each = num_repeats)
)

# Plot
mean_linear <- mean(results_df$Error[results_df$Model == "Linear Effects"])
mean_dummy <- mean(results_df$Error[results_df$Model == "Dummy Encoding"])

ggplot(results_df, aes(x = Error, fill = Model)) +
  geom_density(alpha = 0.5) +
  geom_vline(aes(xintercept = mean_linear, color = "Linear Model Mean"),
             linetype = "dashed", size = 0.5) +
  geom_vline(aes(xintercept = mean_dummy, color = "Dummy Model Mean"),
             linetype = "dashed", size = 0.5) +
  labs(title = "Empirical Distribution of Test Errors",
       x = "Test Error (MSE)",
       y = "Density") +
  scale_color_manual(values = c("Linear Model Mean" = "blue",
                                "Dummy Model Mean" = "red")) +
  theme_light() +
  theme(panel.grid = element_blank(),
        plot.title = element_text(hjust = 0.5),
       text = element_text(size = 10))
```

The graph presents a density plot comparing the test error distributions for the two models. The aim should be illustrate the performance of the models on unseen data and their reliability, focusing on the variability and central tendency of their respective errors. The Linear Model exhibits a narrower distribution centered around a lower MSE compared to the Dummy Model, whose distribution appears wider and shifted slightly

to the right, indicating higer average test error. Basically, the tigter distribution and lower mean MSE of the Linear Model suggest that it generalize better to the test set compared to the dummy model.

Variable Selection Methods in Linear Regression

In this section we will look at model selection, which involves evaluating a trade-off between the goodness of fit of the model, which reflects the ability to fit the observed data, and the complexity of the model, which refers to the number of parameters used. To do this, various model selection criteria have been developed to guide the choice of parsimonious models that offer good generalization. Two of these widely used criteria are the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). The AIC estimates the loss of information when a candidate model is used to represent the real data generation process, aiming to minimize this loss. On the other hand, the BIC is derived from a Bayesian perspective and approaches model selection as choosing the model with the highest posterior probability given the observed data. Both the AIC and BIC take into consideration the goodness of fit of the model and penalize model complexity, although the BIC imposes a stricter penalty for complexity, favoring simpler models than the AIC. The Forward Selection method starts with a model containing only one intercept and iteratively adds variables one at a time, selecting at each step the variable that leads to the maximum reduction in AIC or BIC. In contrast, the Backward method starts with a complete model and iteratively removes variables that contribute less significantly to the model, evaluated again by the AIC or BIC. Both the Stepwise and Backward methods provide systematic strategies for model selection, guided by the goal of finding a model that balances goodness of fit and complexity as quantified by the AIC or BIC.

```r
library(MASS)
library(leaps)

# Backward Elimination
full_model <- lm(LC50 ~ ., data = trainData)
backward_aic <- stepAIC(full_model, direction = "backward",
                        k = 2, trace = FALSE)

backward_bic <- stepAIC(full_model, direction = "backward",
                        k = log(nrow(trainData)), trace = FALSE)
```

```r
summary(backward_aic)
```

```
##
## Call:
## lm(formula = LC50 ~ TPSA + SAacc + MLOGP + RDCHI + GATS1p + nN,
##     data = trainData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.1632 -0.7485 -0.1143  0.6156  3.8169
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.970390   0.278237  10.676  < 2e-16 ***
## TPSA         0.026358   0.003125   8.435 8.35e-16 ***
## SAacc       -0.011638   0.001946  -5.980 5.40e-09 ***
## MLOGP        0.487552   0.073711   6.614 1.37e-10 ***
## RDCHI        0.290699   0.162996   1.783 0.075358 .
## GATS1p      -0.573581   0.179110  -3.202 0.001485 **
## nN          -0.197504   0.054796  -3.604 0.000357 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 1.187 on 357 degrees of freedom
## Multiple R-squared:  0.4707, Adjusted R-squared:  0.4618
## F-statistic: 52.91 on 6 and 357 DF,  p-value: < 2.2e-16
```

```
summary(backward_bic)
```

```
##
## Call:
## lm(formula = LC50 ~ TPSA + SAacc + MLOGP + GATS1p + nN, data = trainData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.0473 -0.7751 -0.0928  0.5879  3.7865
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.123831   0.265406  11.770  < 2e-16 ***
## TPSA         0.028625   0.002863   9.997  < 2e-16 ***
## SAacc       -0.010068   0.001741  -5.783  1.6e-08 ***
## MLOGP        0.594639   0.042886  13.866  < 2e-16 ***
## GATS1p      -0.466421   0.169244  -2.756 0.006152 **
## nN          -0.187790   0.054690  -3.434 0.000665 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.19 on 358 degrees of freedom
## Multiple R-squared:  0.466,  Adjusted R-squared:  0.4585
## F-statistic: 62.47 on 5 and 358 DF,  p-value: < 2.2e-16
```

```
# Forward Selection
null_model <- lm(LC50 ~ 1, data = trainData)
forward_aic <- stepAIC(null_model, direction = "forward",
                       scope = formula(full_model), k = 2, trace = FALSE)

forward_bic <- stepAIC(null_model, direction = "forward",
                       scope = formula(full_model), k = log(nrow(trainData)),
                       trace = FALSE)
```

```
summary(forward_aic)
```

```
##
## Call:
## lm(formula = LC50 ~ MLOGP + TPSA + SAacc + nN + GATS1p + RDCHI,
##     data = trainData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.1632 -0.7485 -0.1143  0.6156  3.8169
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.970390   0.278237  10.676  < 2e-16 ***
## MLOGP        0.487552   0.073711   6.614 1.37e-10 ***
## TPSA         0.026358   0.003125   8.435 8.35e-16 ***
## SAacc       -0.011638   0.001946  -5.980 5.40e-09 ***
```

```
## nN          -0.197504   0.054796  -3.604 0.000357 ***
## GATS1p      -0.573581   0.179110  -3.202 0.001485 **
## RDCHI        0.290699   0.162996   1.783 0.075358 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.187 on 357 degrees of freedom
## Multiple R-squared:  0.4707, Adjusted R-squared:  0.4618
## F-statistic: 52.91 on 6 and 357 DF,  p-value: < 2.2e-16
```

```
summary(forward_bic)
```

```
##
## Call:
## lm(formula = LC50 ~ MLOGP + TPSA + SAacc + nN + GATS1p, data = trainData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.0473 -0.7751 -0.0928  0.5879  3.7865
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.123831   0.265406  11.770  < 2e-16 ***
## MLOGP        0.594639   0.042886  13.866  < 2e-16 ***
## TPSA         0.028625   0.002863   9.997  < 2e-16 ***
## SAacc       -0.010068   0.001741  -5.783  1.6e-08 ***
## nN          -0.187790   0.054690  -3.434 0.000665 ***
## GATS1p      -0.466421   0.169244  -2.756 0.006152 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.19 on 358 degrees of freedom
## Multiple R-squared:  0.466,  Adjusted R-squared:  0.4585
## F-statistic: 62.47 on 5 and 358 DF,  p-value: < 2.2e-16
```

```
# Compare Models
print(backward_aic$call)
```

lm(formula = LC50 ~ TPSA + SAacc + MLOGP + RDCHI + GATS1p + nN, data = trainData)

```
print(backward_bic$call)
```

lm(formula = LC50 ~ TPSA + SAacc + MLOGP + GATS1p + nN, data = trainData)

```
print(forward_aic$call)
```

lm(formula = LC50 ~ MLOGP + TPSA + SAacc + nN + GATS1p + RDCHI, data = trainData)

```
print(forward_bic$call)
```

lm(formula = LC50 ~ MLOGP + TPSA + SAacc + nN + GATS1p, data = trainData)

```
# Model Comparisons
cat(paste(
  "AIC Backward:", AIC(backward_aic), "\n",
  "AIC Forward:", AIC(forward_aic), "\n",
  "BIC Backward:", BIC(backward_bic), "\n",
  "BIC Forward:", BIC(forward_bic), "\n"
```

```
))
```

```
## AIC Backward: 1166.42136187616
##  AIC Forward: 1166.42136187616
##  BIC Backward: 1194.93024571721
##  BIC Forward: 1194.93024571721
```

Both the AIC and BIC values are identical between the forward and backward procedures, suggesting that the model selection methods converge on the same models based on the respective criterion. The AIC favors slightly more complex models by retaining RDCHI, while the BIC prefers more parsimonious models by excluding it.

Ridge Regression Model Optimization

Now we apply Ridge regression to predict the target variable and utilize two methods: cross-validation and bootstrapping to determine the optimal regularization parameter λ. We are doing this because Ridge regression is particularly useful in reducing model complexity and handling multicollinearity by penalizing large coefficients and could improve our current situation. We explore a grid of λ values and contrast the outcomes of the two methods, each aimed at finding the best balance between bias and variance.

The cross-validation method used is 10-fold and the bootstrap procedure resample the training data 100 times. Both techniques are applied to a manually chosen grid of candidate λ values ranging from 10^3 to 10^{-2}.

```
library(glmnet)
library(boot)

index <- sample(seq_len(nrow(data)), size = 2/3 * nrow(data))
trainData <- data[index, ]
testData <- data[-index, ]

X_train <- as.matrix(trainData[, -ncol(trainData)])
y_train <- trainData$LC50
X_test <- as.matrix(testData[, -ncol(testData)])
y_test <- testData$LC50

lambda_grid <- 10^seq(3, -2, lenght = 100)

# Ridge regression (CV)
cv_ridge <- cv.glmnet(X_train, y_train, alpha = 0,
                      lambda = lambda_grid, nfolds = 10)
optimal_lambda_cv <- cv_ridge$lambda.min
```

```
cat("Optimal lambda from Cross-Validation:", optimal_lambda_cv, "\n")
```

```
## Optimal lambda from Cross-Validation: 0.01
```

```
ridge_model <- glmnet(X_train, y_train, alpha = 0,
                      lambda = optimal_lambda_cv)

train_predictions_ridge <- predict(ridge_model, newx = X_train)
test_predictions_ridge <- predict(ridge_model, newx = X_test)

mse_train_ridge <- mean((y_train - train_predictions_ridge)^2)
mse_test_ridge <- mean((y_test - test_predictions_ridge)^2)
```

```
cat(paste(
  "Training Error (Ridge):", mse_train_ridge, "\n",
```

```
  "Test Error (Ridge):", mse_test_ridge, "\n"))
```

```
## Training Error (Ridge): 1.37728208372072
##  Test Error (Ridge): 1.5540291520742
```

The cross-validation process identifies the optimal λ by splitting the dataset into 10 folds and measuring the model's performance on different subsets. This approach basically tends to balance bias and variance by repeatedly testing the model on different parts of the data. The result showed an optimal λ of 0.01 with a corresponding training error (MSE) of 1.377282 and a test error of 1.554029, indicating that the model performs well on both the training and test sets, demonstrating low variance and good generalization.

```
# Ridge regression (bootstrap)
bootstrap_mse <- function(data, indices, lambda) {
  # Create sample
  bootstrap_sample <- data[indices, ]
  X_bootstrap <- as.matrix(bootstrap_sample[, -ncol(bootstrap_sample)])
  y_bootstrap <- bootstrap_sample$LC50

  model <- glmnet(X_bootstrap, y_bootstrap, alpha = 0, lambda = lambda)
  y_pred <- predict(model, s = lambda, newx = X_test)
  mse <- mean((y_test - y_pred)^2)
  return(mse)
}
# Bootstrap for multiple lambda values
bootstrap_results <- sapply(lambda_grid, function(lambda) {
  mse_values <- replicate(100, boot(trainData, bootstrap_mse, R = 1,
                                    lambda = lambda)$t)
  return(mean(mse_values))
})
```

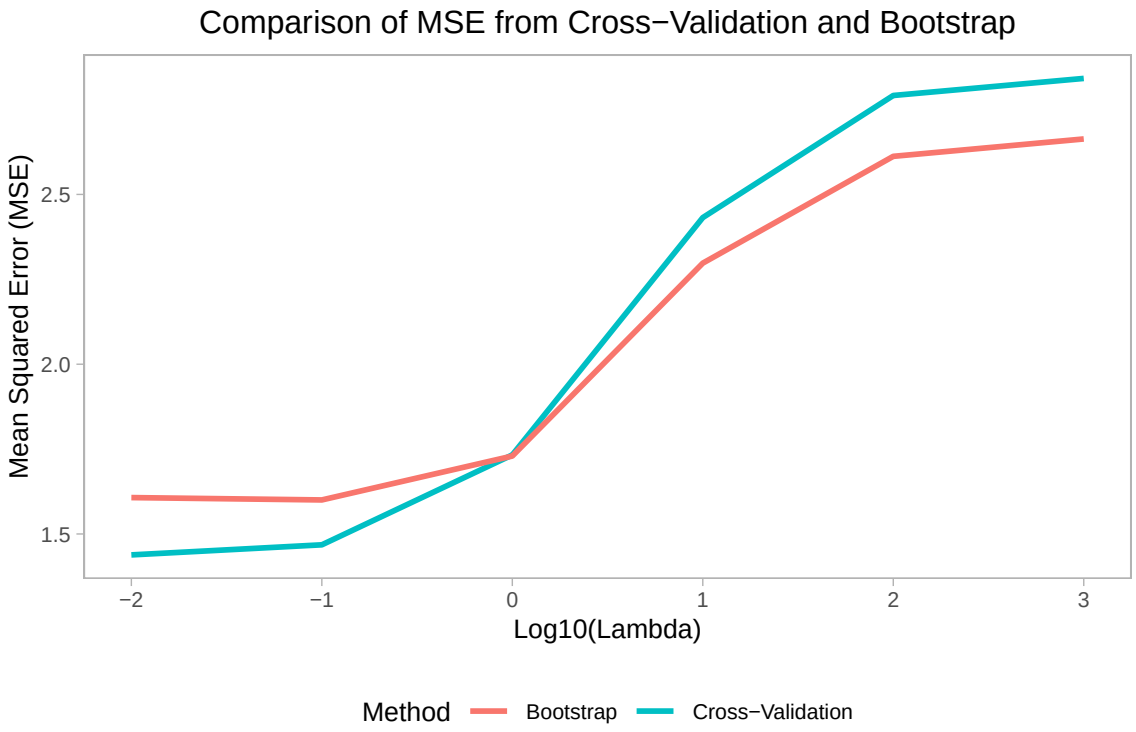
```
# Optimal Lambda from Bootstrap
optimal_lambda_bootstrap <- lambda_grid[which.min(bootstrap_results)]
cat("Optimal lambda from Bootstrap:", optimal_lambda_bootstrap, "\n")
```

```
## Optimal lambda from Bootstrap: 0.1
```

The bootstrap method resamples the dataset 100 times to evaluate the stability of the model and its performance under various random samples. The key advantage of this approach is that it generates different training datasets to capture variability. The bootstrap results across multiple λ values yielded an optimal λ of 0.1.

```
results_df <- data.frame(
  Lambda = lambda_grid,
  CV_MSE = sapply(lambda_grid,
                  function(l) mean(cv_ridge$cvm[cv_ridge$lambda == l])),
  Bootstrap_MSE = bootstrap_results
)

# Plot
ggplot(results_df, aes(x = log10(Lambda)) ) +
  geom_line(aes(y = CV_MSE, color = "Cross-Validation"), size = 1) +
  geom_line(aes(y = Bootstrap_MSE, color = "Bootstrap"), size = 1) +
  labs(title = "Comparison of MSE from Cross-Validation and Bootstrap",
       x = "Log10(Lambda)",
       y = "Mean Squared Error (MSE)",
       color = "Method") +
```

```
  theme_light() +
  theme(legend.position = "bottom",
        panel.grid = element_blank(),
        plot.title = element_text(hjust = 0.5),
        text = element_text(size = 10))
```



The results of the two methods are contrasted using a plot, where the MSE curves are plotted against $\log_{10}(\lambda)$. Both curves exhibit a similar trend, showing a U-shape where error decreases initially with decreasing λ (left-hand side of the plot) before rising as λ increases.

The optimal λ for cross-validation is slightly lower ($\log_{10}(\lambda) = -2$) compared to that of bootstrap ($\log_{10}(\lambda) = -1$). The Cross-Validation method may prefer a model with more regularization compared to Bootstrap, which tends to allow slightly more complexity (i.e., a higher λ). So in this case, Cross-Validation seems to provide a more stable and slightly better performance estimate, as shown by the lower MSE values.

Non-linear Modeling Using Generalized Additive Models

Generalized Additive Models (GAMs) offer a useful compromise between linear models and fully non-parametric models. They allow fitting a non-linear function f_j to each X_j, enabling the automatic modeling of non-linear relationships that standard linear regression would not detect. Since the model is additive, we can still examine the effect of each X_j on Y individually, keeping all other variables fixed. The smoothness of the function f_j for the variable X_j can be summarized by the degrees of freedom. The main limitation of GAMs is that the model is constrained to be additive. With many variables, it is possible to miss important interactions. However, as with linear regression, we can manually add interaction terms to the GAM model by including additional predictors in the form of $X_j \times X_k$. We can also add low-dimensional interaction functions in the form of $f_{jk}(X_j, X_k)$ to the model. Such terms can be fitted using two-dimensional smoothers, such as local regression or two-dimensional splines.

```
sapply(trainData, function(x) length(unique(x)))
```

```
##   TPSA  SAacc   H050  MLOGP  RDCHI GATS1p     nN   C040   LC50
##    171    160      9    283    259    289      9      6    347
```

```
# GAM less complexity (k = -1)
k = 1
gam_model_1 <- gam(LC50 ~ s(TPSA, k=k) + s(SAacc, k=k) + s(H050, k=k) +
                     s(MLOGP, k=k) + s(RDCHI, k=k) + s(GATS1p, k=k) +
                     s(nN, k=k) + s(C040, k=k), data = trainData)

pred_gam_train_1 <- predict(gam_model_1, newdata = trainData)
pred_gam_test_1 <- predict(gam_model_1, newdata = testData)
mse_train_gam_1 <- mean((y_train - pred_gam_train_1)^2)
mse_test_gam_1 <- mean((y_test - pred_gam_test_1)^2)

cat(paste(
  "Training Error (GAM - k=1):","\t",mse_train_gam_1, "\n",
  "Test Error (GAM - k=1):","\t",mse_test_gam_1, "\n"))
```

```
## Training Error (GAM - k=1):   1.30000499663681
##  Test Error (GAM - k=1):      1.56539408308404
```

```
# GAM more complexity (k = 6)
k = 6
gam_model_2 <- gam(LC50 ~ s(TPSA, k=k) + s(SAacc, k=k) + s(H050, k=k) +
                     s(MLOGP, k=k) + s(RDCHI, k=k) + s(GATS1p, k=k) +
                     s(nN, k=k) + s(C040, k=k), data = trainData)

pred_gam_train_2 <- predict(gam_model_2, newdata = trainData)
pred_gam_test_2 <- predict(gam_model_2, newdata = testData)
mse_train_gam_2 <- mean((y_train - pred_gam_train_2)^2)
mse_test_gam_2 <- mean((y_test - pred_gam_test_2)^2)

cat(paste(
  "Training Error (GAM - k=6):","\t",mse_train_gam_2, "\n",
  "Test Error (GAM - k=6):","\t",mse_test_gam_2, "\n"))
```

```
## Training Error (GAM - k=6):   1.14788267393574
##  Test Error (GAM - k=6):      1.63477459427387
```

The first model (`gam_model_1`) is specified with $k = 1$, indicating a lower complexity where each smooth term is linear. The second model (`gam_model_2`) is specified with $k = 6$ (higher value for the model due to unique values for each variable), allowing for more flexibility and complexity in the smooth terms.

gam_model_2 has a lower training error, its increased complexity results in higher test error compared to gam_model_1. Therefore, despite gam_model_1's higher test error, it may be preferable due to its generalization capability.

Regression Tree Model with Cost-Complexity Pruning

Now, we use a regression tree model to predict the response variable based. For regression tasks, the ANOVA split is employed, where splits are made to minimize the variance within nodes, effectively partitioning the data into homogeneous subsets. Regression trees operate by recursively splitting the data along the most significant feature at each node, creating a model that makes piecewise constant predictions for the response variable. While decision trees are highly flexible and can capture complex interactions in the data, they are prone to overfitting, particularly when the tree grows too deep. To mitigate this, pruning or controlling the complexity of the tree with a complexity parameter (CP) is often applied, which helps prevent overfitting by stopping splits that provide minimal improvement. Here, we implement a regression tree using the `rpart`

library and assess the model's performance by comparing the Mean Squared Error (MSE) on both training and test datasets.

```
library(rpart)
library(rpart.plot)

tree_model <- rpart(LC50 ~ ., data = trainData, method = "anova")

train_predictions_tree1 <- predict(tree_model, newdata = trainData)
test_predictions_tree1 <- predict(tree_model, newdata = testData)

mse_train_tree1 <- mean((trainData$LC50 - train_predictions_tree1)^2)
mse_test_tree1 <- mean((testData$LC50 - test_predictions_tree1)^2)
```

The complexity parameter (CP) table presented in the output illustrates the impact of increasing tree complexity by adding splits. The root node error serves as the baseline error when no splits are made. As splits are introduced, the relative error decreases, with the corresponding cross-validated error (xerror) reflecting how well the model is expected to generalize to unseen data.

```
printcp(tree_model)
```

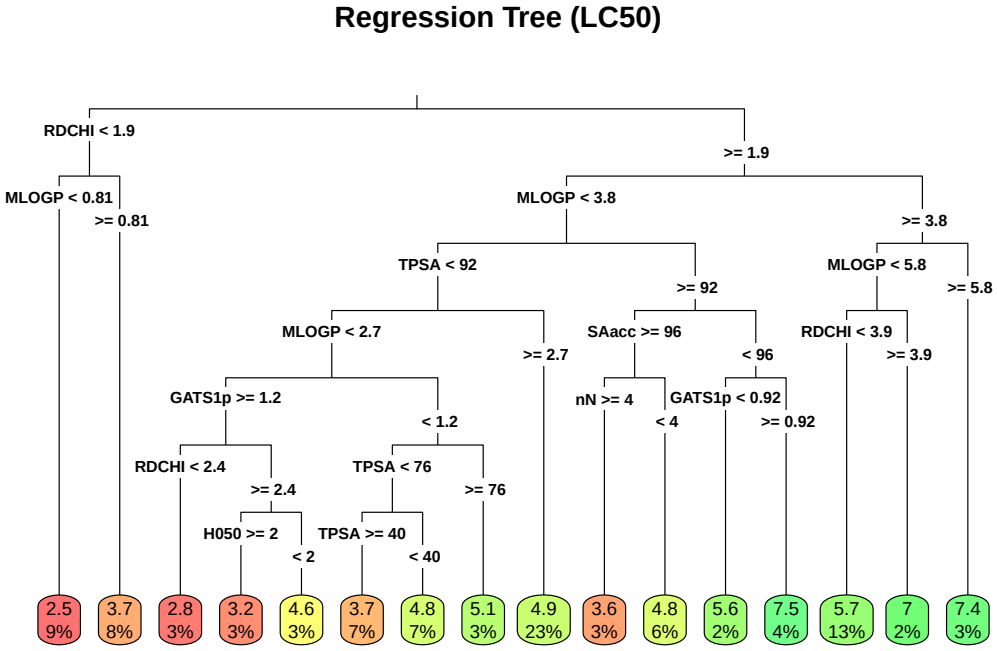
```
##
## Regression tree:
## rpart(formula = LC50 ~ ., data = trainData, method = "anova")
##
## Variables actually used in tree construction:
## [1] GATS1p H050   MLOGP  nN     RDCHI  SAacc  TPSA
##
## Root node error: 949.5/364 = 2.6085
##
## n= 364
##
##          CP nsplit rel error  xerror     xstd
## 1  0.202576      0   1.00000 1.00521 0.081444
## 2  0.120003      1   0.79742 0.89217 0.074688
## 3  0.055504      2   0.67742 0.71621 0.060411
## 4  0.029750      4   0.56641 0.68528 0.057679
## 5  0.024807      5   0.53666 0.69253 0.060911
## 6  0.019644      6   0.51186 0.69168 0.061473
## 7  0.018758      7   0.49221 0.68003 0.061525
## 8  0.018127      8   0.47345 0.68003 0.061525
## 9  0.012598      9   0.45533 0.68531 0.063366
## 10 0.011999     10   0.44273 0.70330 0.069257
## 11 0.011431     11   0.43073 0.70041 0.069312
## 12 0.011033     13   0.40787 0.69407 0.069352
## 13 0.010592     14   0.39684 0.69141 0.068352
## 14 0.010000     15   0.38625 0.68556 0.068029
```

```
cat(paste(
  "Training Error (Tree):","\t", mse_train_tree1, "\n",
  "Test Error (Tree):","\t", mse_test_tree1, "\n"))
```

```
## Training Error (Tree):    1.00753081670867
##  Test Error (Tree):   2.053223625723
```

The decision tree model achieved a training error of 1.0075 and a test error of 2.0532, indicating that while the model fits the training data relatively well, it exhibits poor generalization on the test data. This could be

due to potential overfitting, where the model captures noise or spurious relationships within the training data that do not generalize well to new data. In comparison to previously explored models, such as the Generalized Additive Model (GAM) with k=1, which demonstrated better test error performance, the decision tree's higher test error means that further tuning, such as pruning or adjusting the complexity parameter, may be necessary to improve its predictive performance on unseen data.



In this updated approach, we prune the previously trained decision tree to enhance its generalization by reducing model complexity.The optimal complexity parameter (CP) is selected from the tree's complexity table based on the lowest cross-validated error (xerror), ensuring an appropriate balance between model complexity and prediction accuracy. Using this optimal CP value, we apply the prune function to simplify the tree by reducing the number of splits. The results show a training error of 1.2839 and a test error of 2.0162, compared to the unpruned tree's training error of 1.0075 and test error of 2.0532. As expected, the training error increased after pruning because the model is now less complex and less tailored to the training data. However, the test error has decreased slightly, indicating a small improvement in the model's ability to generalize to new data. While the improvement in test error is small, this process is crucial for producing a model that generalizes better without losing too much predictive power. In comparison to the unpruned tree, the pruned version provides a more balanced model by maintaining reasonable predictive performance while preventing overfitting, making it a valuable step in building robust machine learning models.

```
optimal_cp <- tree_model$cptable[which.min(tree_model$cptable[,"xerror"]), "CP"]

pruned_tree <- prune(tree_model, cp = optimal_cp)
train_predictions_tree <- predict(pruned_tree, newdata = trainData)
test_predictions_tree <- predict(pruned_tree, newdata = testData)

mse_train_tree <- mean((trainData$LC50 - train_predictions_tree)^2)
mse_test_tree <- mean((testData$LC50 - test_predictions_tree)^2)
```

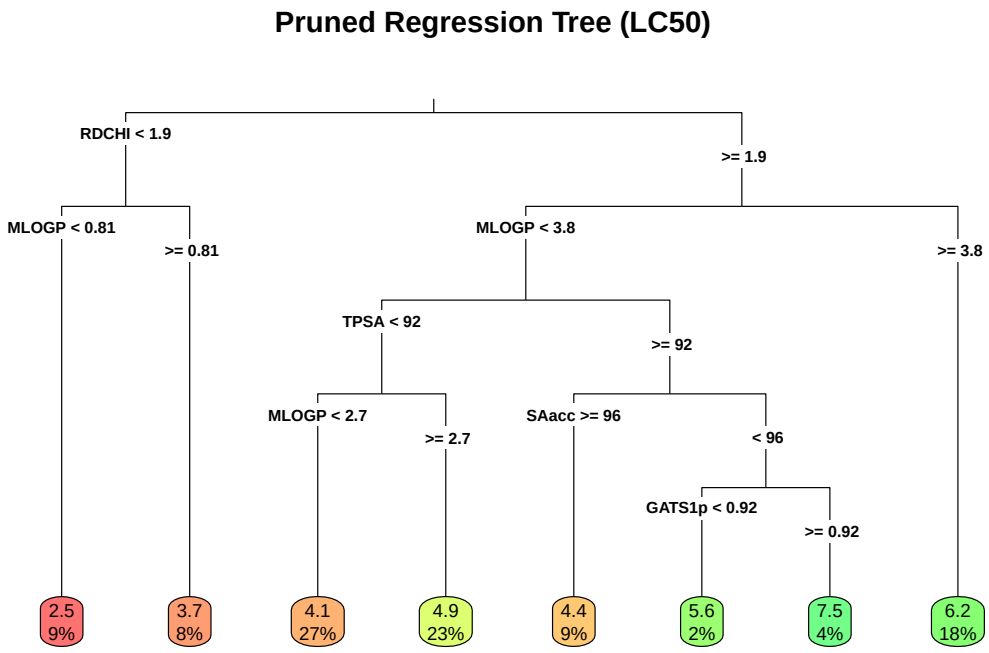
```
printcp(pruned_tree)
```

```
##
```

```
## Regression tree:
## rpart(formula = LC50 ~ ., data = trainData, method = "anova")
##
## Variables actually used in tree construction:
## [1] GATS1p MLOGP  RDCHI  SAacc  TPSA
##
## Root node error: 949.5/364 = 2.6085
##
## n= 364
##
##         CP nsplit rel error  xerror     xstd
## 1 0.202576      0   1.00000 1.00521 0.081444
## 2 0.120003      1   0.79742 0.89217 0.074688
## 3 0.055504      2   0.67742 0.71621 0.060411
## 4 0.029750      4   0.56641 0.68528 0.057679
## 5 0.024807      5   0.53666 0.69253 0.060911
## 6 0.019644      6   0.51186 0.69168 0.061473
## 7 0.018758      7   0.49221 0.68003 0.061525
```

```
cat(paste("Training Error (Tree Pruned):","\t", mse_train_tree, "\n",
          "Test Error (Tree Pruned):","\t", mse_test_tree, "\n"))
```

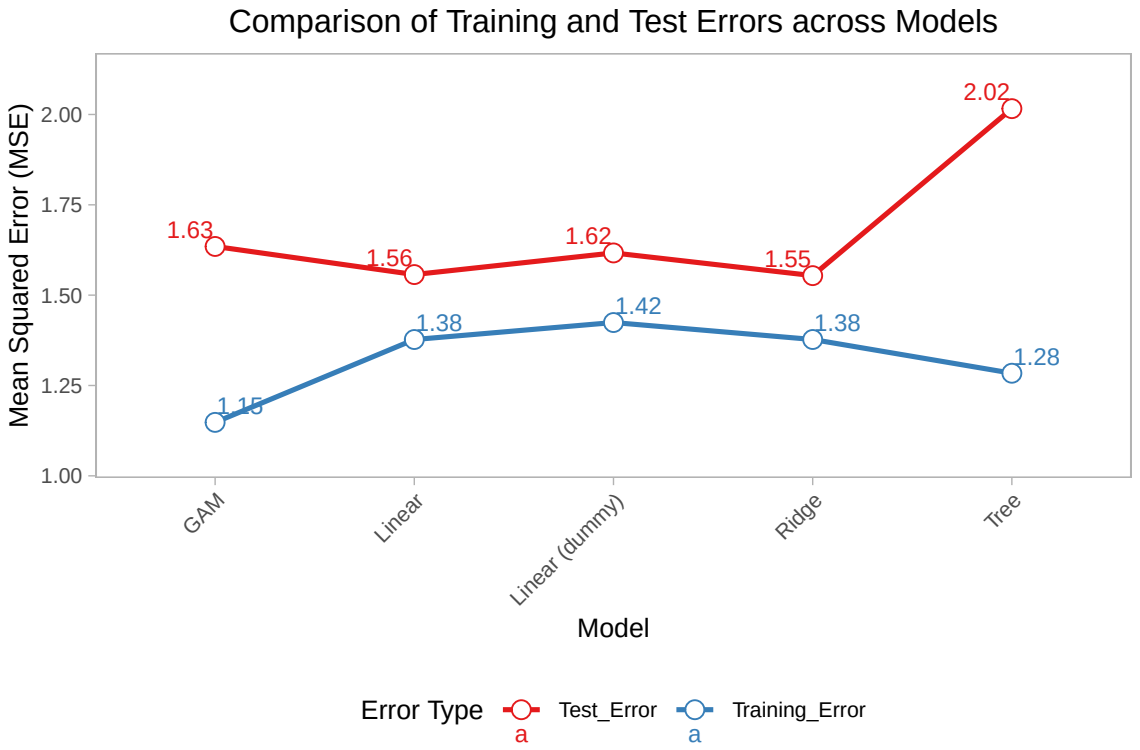
```
## Training Error (Tree Pruned):     1.28394924754904
##  Test Error (Tree Pruned):    2.01615150806028
```



Comparative Analysis of Regression Models

In the final part of this first chapter, we compared the performance of all models we studied. The Mean Squared Error (MSE) for both training and test datasets was calculated for each model, and the results were plotted in the graph above. The plot clearly shows the trade-off between model complexity and generalization.

Models like GAM and Linear Regression exhibit a balance between training and test error, whereas the Decision Tree shows a significant increase in test error, indicating overfitting despite its lower training error. Ridge Regression, on the other hand, strikes a balance with relatively low test error compared to other models, suggesting that regularization helped in avoiding overfitting while maintaining good performance.



From this comparison, we can conclude that Ridge Regression offers the best overall performance in terms of balancing training and test errors. The Decision Tree, while effective in minimizing training error, suffers from poor generalization on unseen data, as reflected by the highest test error.

Classification Model Comparison for Diabetes Prediction

Now in this second chapter, we use the Pima Indians Diabetes Database, a widely recognized dataset for testing and evaluating classification models. The dataset contains information on 768 women from the Pima Indian population, a group known to have a high predisposition to diabetes. The primary goal is to predict whether a woman has developed diabetes based on a set of eight numeric features. The response variable, diabetes, is binary, indicating the presence (`pos`) or absence (`neg`) of the disease. The features include medical and physiological factors such as the number of pregnancies (`pregnant`), plasma glucose concentration (`glucose`), diastolic blood pressure (`pressure`), triceps skin fold thickness (`triceps`), 2-hour serum insulin levels (`insulin`), body mass index (`mass`), diabetes pedigree function (`pedigree`), and `age`.

```
summary(df)
```

```
##     pregnant         glucose         pressure         triceps
##  Min.   : 0.000   Min.   : 44.0   Min.   : 24.00   Min.   : 7.00
##  1st Qu.: 1.000   1st Qu.: 99.0   1st Qu.: 64.00   1st Qu.:22.00
##  Median : 3.000   Median :117.0   Median : 72.00   Median :29.00
##  Mean   : 3.845   Mean   :121.7   Mean   : 72.41   Mean   :29.15
##  3rd Qu.: 6.000   3rd Qu.:141.0   3rd Qu.: 80.00   3rd Qu.:36.00
##  Max.   :17.000   Max.   :199.0   Max.   :122.00   Max.   :99.00
##                   NA's   :5       NA's   :35       NA's   :227
```

```
##     insulin            mass          pedigree           age        diabetes
##  Min.   : 14.00   Min.   :18.20   Min.   :0.0780   Min.   :21.00   neg:500
##  1st Qu.: 76.25   1st Qu.:27.50   1st Qu.:0.2437   1st Qu.:24.00   pos:268
##  Median :125.00   Median :32.30   Median :0.3725   Median :29.00
##  Mean   :155.55   Mean   :32.46   Mean   :0.4719   Mean   :33.24
##  3rd Qu.:190.00   3rd Qu.:36.60   3rd Qu.:0.6262   3rd Qu.:41.00
##  Max.   :846.00   Max.   :67.10   Max.   :2.4200   Max.   :81.00
##  NA's   :374      NA's   :11
```

```
print(na_count)
```

```
## pregnant  glucose pressure  triceps  insulin     mass pedigree      age
##        0        5       35      227      374       11        0        0
## diabetes
##        0
```

The first thing that we do is look at the NA's values. We address this problem by employing a k-Nearest Neighbors (kNN) imputation method. Missing data can lead to biased model results and reduced predictive performance, so handling these gaps appropriately is crucial. We observe that several features, such as glucose, pressure, triceps, insulin, and mass, contain missing values, as indicated by the na_count summary. The kNN imputation approach is implemented to estimate and fill these missing values.

```
# KNN imputation
library(mlbench)
library(VIM)

df_imp <- kNN(df, k = 5)
df_imp <- df_imp[, 1:9]
df_imp$diabetes <- as.factor(df_imp$diabetes)
na_count_df_imp <- sapply(df_imp, function(x) sum(is.na(x)))
print(na_count_df_imp)
```

```
## pregnant  glucose pressure  triceps  insulin     mass pedigree      age
##        0        0        0        0        0        0        0        0
## diabetes
##        0
```

`kNN imputation` is a popular non-parametric method for handling missing data. The idea behind this method is to use the nearest data points to estimate missing values by leveraging the similarity between observations in the dataset. The underlying assumption is that similar instances will have similar values for the missing features. In a typical kNN algorithm, the distance between data points is calculated based on the non-missing features. Once the k-nearest neighbors (with the closest distance) to a given data point are identified, the missing values are imputed based on some aggregation (e.g., mean, median, or mode) of the nearest neighbors' corresponding values.

k-NN Classification and Cross-Validation Comparison

```
library(caret)
set.seed(2024)
trainIndex <- createDataPartition(df_imp$diabetes,
                                  p = 2/3,
                                  list = FALSE,
                                  times = 1)

trainData <- df_imp[trainIndex, ]
testData <- df_imp[-trainIndex, ]
```

```
trainData$diabetes <- ifelse(trainData$diabetes == "pos", 1, 0)
testData$diabetes <- ifelse(testData$diabetes == "pos", 1, 0)
trainData$diabetes <- as.factor(trainData$diabetes)
testData$diabetes <- as.factor(testData$diabetes)
```

To evaluate the performance of the k-Nearest Neighbors (k-NN) classifier on the diabetes dataset, we conducted a systematic investigation into the optimal number of neighbors, k, by employing both 5-fold cross-validation and leave-one-out cross-validation (`LOOCV`). The analysis involved calculating the error rates across a range of k values (from 1 to 25), allowing us to assess how the choice of k influences model accuracy.

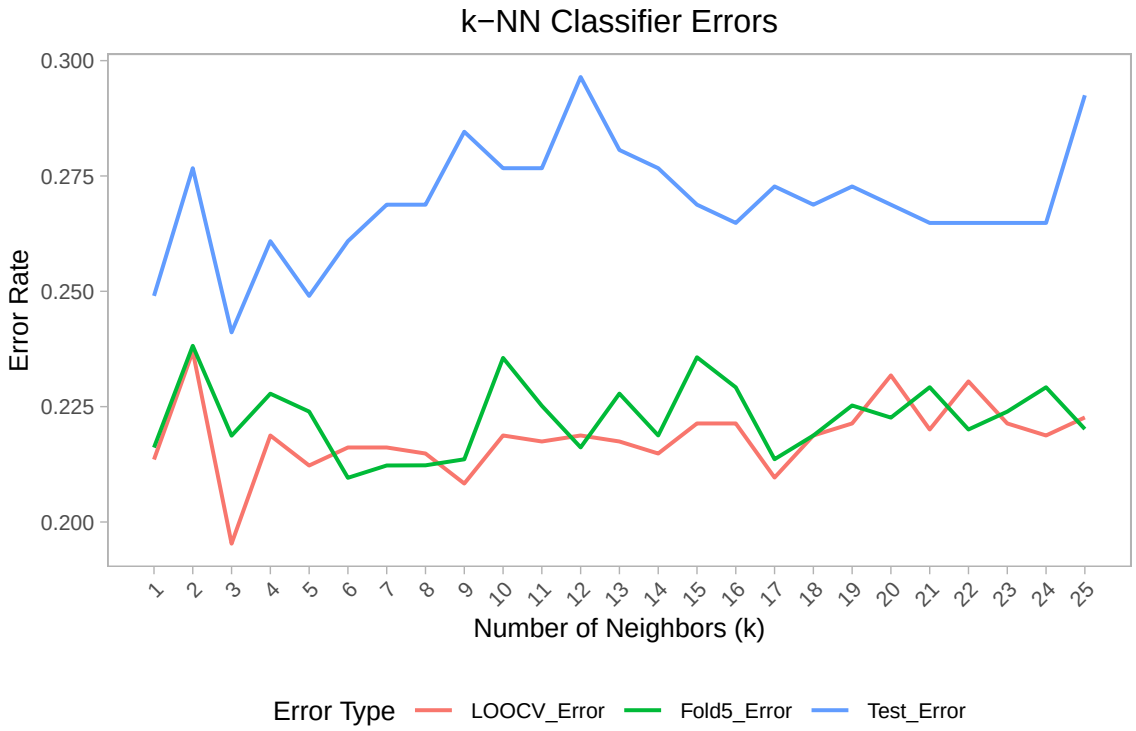
```
library(class)
library(mlbench)

cv_knn_errors <- function(data, k_values, cv_type) {
    errors <- numeric(length(k_values))
    for (i in seq_along(k_values)) {
        k <- k_values[i]
        if (cv_type == "LOOCV") {
            control <- trainControl(method = "LOOCV")
        } else if (cv_type == "5-fold") {
            control <- trainControl(method = "cv", number = 5)
        }

        model <- train(diabetes ~ ., data = data, method = "knn",
                       trControl = control, tuneGrid = data.frame(k = k))
        errors[i] <- min(model$results$Accuracy)
    }
    return(1 - errors)
}

# k values (tested that >25 is too much)
k_values <- 1:25

# 5 fold
errors_5fold <- cv_knn_errors(df_imp, k_values, "5-fold")

# LOOCV
errors_loocv <- cv_knn_errors(df_imp, k_values, "LOOCV")

trainIndex <- createDataPartition(df_imp$diabetes, p = .67, list = FALSE)
trainData <- df_imp[trainIndex, ]
testData <- df_imp[-trainIndex, ]

# Loop for k
test_errors <- numeric(length(k_values))
for (i in seq_along(k_values)) {
    k <- k_values[i]
    predictions <- knn(train = trainData[,-ncol(trainData)],
                       test = testData[,-ncol(testData)],
                       cl = trainData$diabetes, k = k)
    test_errors[i] <- mean(predictions != testData$diabetes)
}
optimal_k_5fold <- k_values[which.min(errors_5fold)]
optimal_k_loocv <- k_values[which.min(errors_loocv)]
```

```r
optimal_k_test <- k_values[which.min(test_errors)]
```

```r
cat(paste("Optimal k from 5-fold CV:", optimal_k_5fold, "\n",
          "Optimal k from LOOCV:", optimal_k_loocv, "\n",
          "Optimal k from Test Errors:", optimal_k_test, "\n"))
```

```
## Optimal k from 5-fold CV: 6
##  Optimal k from LOOCV: 3
##  Optimal k from Test Errors: 3
```

The optimal k determined through 5-fold cross-validation was 6, while both leave-one-out cross-validation (LOOCV) and test errors indicated an optimal k of 3. These findings suggest that while a slightly larger neighborhood can enhance accuracy in cross-validation settings, the model may perform best with a smaller k when generalizing to unseen data. This discrepancy underscores the inherent trade-offs between bias and variance in model selection, as a lower k can lead to greater sensitivity to noise in the training data, potentially improving predictive performance on test datasets.



The blue test error line shows noticeable variability across different k values, indicating that the model's ability to generalize decreases as k increases. Both LOOCV and 5-fold cross-validation errors remain relatively stable. These cross-validation methods offer consistent error estimates regardless of small changes in k. The optimal k for the test error seems to be around k=3, where the test error reaches its minimum, and both cross-validation errors align closely, validating this selection.

```r
k_value <- 3
predictions <- knn(train = trainData[,-ncol(trainData)],
                   test = testData[,-ncol(testData)],
                   cl = trainData$diabetes,
                   k = k_value)

confusion_matrix <- table(testData$diabetes, predictions)
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
```

```r
print(confusion_matrix)
```

```
##      predictions
##       neg pos
##   neg 129  36
##   pos  25  63
```

```r
cat("Accuracy of k-NN with k =", k_value, ":", accuracy, "\n")
```

```
## Accuracy of k-NN with k = 3 : 0.7588933
```

The confusion matrix for `k=3` shows that the model correctly classified 129 negative cases and 63 positive cases, but misclassified 25 positive cases as negative and 36 negative cases as positive. This leads to an overall accuracy of 75.89%.

Generalized Additive Model for Diabetes Prediction

In this section, we look again at Generalized Additive Model (GAM) but now using splines and applying variable selection to identify the best model. GAMs allow for flexible modeling by applying smooth functions (splines) to predictor variables, capturing non-linear relationships without requiring specific parametric forms.

We first fit a GAM using smooth functions `s()` on each predictor in the diabetes dataset. The response variable, diabetes, is binary, so we use the binomial family with the logit link function.

```r
library(caret)
library(mgcv)
library(dplyr)
library(broom)

# model
gam_model <- gam(diabetes ~ s(pregnant) + s(glucose) + s(pressure) +
                 s(triceps) + s(insulin) + s(mass) +
                 s(pedigree) + s(age), data = trainData, family = binomial)

pred_probs <- predict(gam_model, newdata = testData, type = "response")
pred_class <- ifelse(pred_probs > 0.5, "pos", "neg") # Threshold at 0.5

# Confusion Matrix
conf_matrix <- confusionMatrix(as.factor(pred_class), testData$diabetes, positive = "pos")
```

```r
summary(gam_model)
```

```
##
## Family: binomial
## Link function: logit
##
## Formula:
## diabetes ~ s(pregnant) + s(glucose) + s(pressure) + s(triceps) +
##     s(insulin) + s(mass) + s(pedigree) + s(age)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.1915     0.1608  -7.408 1.28e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
```

```
##                edf Ref.df Chi.sq  p-value
## s(pregnant) 1.000  1.000  2.178 0.140000
## s(glucose)  2.524  3.205 19.290 0.000324 ***
## s(pressure) 1.000  1.000  0.299 0.584925
## s(triceps)  2.105  2.708  2.834 0.333325
## s(insulin)  4.204  5.185 29.628 2.68e-05 ***
## s(mass)     3.545  4.466  9.896 0.057375 .
## s(pedigree) 1.000  1.000  6.274 0.012256 *
## s(age)      2.677  3.355 15.802 0.002041 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.442   Deviance explained = 40.3%
## UBRE = -0.15306  Scale est. = 1         n = 515
```

```
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 135  24
##        pos  30  64
##
##                Accuracy : 0.7866
##                  95% CI : (0.7309, 0.8354)
##     No Information Rate : 0.6522
##     P-Value [Acc > NIR] : 2.205e-06
##
##                   Kappa : 0.5369
##
##  Mcnemar's Test P-Value : 0.4962
##
##             Sensitivity : 0.7273
##             Specificity : 0.8182
##          Pos Pred Value : 0.6809
##          Neg Pred Value : 0.8491
##              Prevalence : 0.3478
##          Detection Rate : 0.2530
##    Detection Prevalence : 0.3715
##       Balanced Accuracy : 0.7727
##
##        'Positive' Class : pos
##
```

To find the best model, we can gradually remove less significant predictors based on their p-values from the summary of the GAM. This process reduces the model complexity while retaining important variables. The second GAM model removes less significant predictors `pregnant, pressure, triceps` and keeps the most relevant ones `glucose, insulin, mass, pedigree, age`, based on their smooth term significance.

```
gam_model2 <- gam(diabetes ~ s(glucose) + s(insulin) + s(mass) +
                  s(pedigree) + s(age), data = trainData, family = binomial)

# Make predictions on the test set
pred_probs <- predict(gam_model2, newdata = testData, type = "response")
pred_class <- ifelse(pred_probs > 0.5, "pos", "neg") # Threshold at 0.5
```

```r
# Confusion Matrix
conf_matrix <- confusionMatrix(as.factor(pred_class), testData$diabetes, positive = "pos")
```

```r
summary(gam_model2)
```

```
##
## Family: binomial
## Link function: logit
##
## Formula:
## diabetes ~ s(glucose) + s(insulin) + s(mass) + s(pedigree) +
##     s(age)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.1817     0.1574  -7.509 5.98e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##               edf Ref.df Chi.sq  p-value
## s(glucose)  2.015  2.565  18.25 0.000238 ***
## s(insulin)  4.259  5.255  30.55 1.95e-05 ***
## s(mass)     3.518  4.442  12.90 0.017025 *
## s(pedigree) 1.000  1.000   5.60 0.017975 *
## s(age)      2.830  3.519  32.54 1.68e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.438   Deviance explained = 39.2%
## UBRE = -0.15587  Scale est. = 1         n = 515
```

```r
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 135  22
##        pos  30  66
##
##                Accuracy : 0.7945
##                  95% CI : (0.7394, 0.8425)
##     No Information Rate : 0.6522
##     P-Value [Acc > NIR] : 5.283e-07
##
##                   Kappa : 0.5564
##
##  Mcnemar's Test P-Value : 0.3317
##
##             Sensitivity : 0.7500
##             Specificity : 0.8182
##          Pos Pred Value : 0.6875
##          Neg Pred Value : 0.8599
```

```
##              Prevalence : 0.3478
##          Detection Rate : 0.2609
##    Detection Prevalence : 0.3794
##       Balanced Accuracy : 0.7841
##
##        'Positive' Class : pos
##
```

The full model achieved an accuracy of 78.66%, with a sensitivity of 72.73% and a specificity of 81.82%. It correctly classified 135 non-diabetic cases and 63 diabetic cases, but misclassified 24 diabetic individuals as negative and 30 non-diabetic individuals as positive. The reduced model, which included only a subset of the original predictors, showed a slightly improved performance. It achieved an accuracy of 79.45%, with a sensitivity of 75.00% and a specificity of 81.82%. This model reduced the false negatives from 24 to 22, meaning it more effectively identified positive diabetes cases, while maintaining the same number of false positives (30).

Tree-Based Methods for Diabetes Classification

In this section, we fit three tree-based models: a classification tree, an ensemble of bagged trees and a random forest. The performance of these models is evaluated by comparing the training and test error rates.

The classification tree, had a training error of 0.136 and a test error of 0.217. This result highlights that the model may have slightly overfitted the training data, as it performed better on the training set compared to the test set. The relatively higher test error means that the classification tree is more prone to overfitting compared to ensemble methods.

```
library(rpart)
library(ipred)
library(randomForest)
library(caret)

classification_tree <- rpart(diabetes ~ ., data = trainData, method = "class")

tree_train_pred <- predict(classification_tree, trainData, type = "class")
tree_test_pred <- predict(classification_tree, testData, type = "class")

# error
tree_train_error <- mean(tree_train_pred != trainData$diabetes)
tree_test_error <- mean(tree_test_pred != testData$diabetes)
```

```
cat(paste(
  "Classification Tree Training Error:", tree_train_error,"\n",
  "Classification Tree Test Error:", tree_test_error,"\n"))
```

```
## Classification Tree Training Error: 0.133980582524272
##  Classification Tree Test Error: 0.217391304347826
```

The bagged trees, which is an ensemble of multiple trees created by bootstrapping, exhibited an exceptionally low training error of 0.002, the model perfectly fit the training data. However, the test error was 0.237, showing a significant increase from the training error. This sharp contrast between training and test errors indicates that the bagged model might be overfitting, as it achieves almost perfect performance on the training data but struggles to generalize to unseen data.

```
set.seed(2024)
bagged_trees <- bagging(diabetes ~ ., data = trainData)

bagged_train_pred <- predict(bagged_trees, trainData)
```

```r
bagged_test_pred <- predict(bagged_trees, testData)

# error
bagged_train_error <- mean(bagged_train_pred != trainData$diabetes)
bagged_test_error <- mean(bagged_test_pred != testData$diabetes)
```

```r
cat(paste(
  "Bagged Trees Training Error:", bagged_train_error, "\n",
  "Bagged Trees Test Error:", bagged_test_error, "\n"))
```

```
## Bagged Trees Training Error: 0.00194174757281553
##  Bagged Trees Test Error: 0.237154150197628
```

The random forest, another ensemble method but adds randomness by selecting a subset of features at each split, had a training error of 0 indicating perfect fit on the training data. The test error for the random forest model was 0.217, identical to that of the classification tree. Despite the zero training error, the random forest did not show a marked improvement in the test error, suggesting that while it captures patterns in the training data well, it may still struggle with generalization.

```r
# Fit the random forest
set.seed(2024)
random_forest <- randomForest(diabetes ~ ., data = trainData)

# Predict on training and test data
rf_train_pred <- predict(random_forest, trainData)
rf_test_pred <- predict(random_forest, testData)

# Calculate training and test error
rf_train_error <- mean(rf_train_pred != trainData$diabetes)
rf_test_error <- mean(rf_test_pred != testData$diabetes)
```

```r
cat(paste(
  "Random Forest Training Error:", rf_train_error, "\n",
  "Random Forest Test Error:", rf_test_error, "\n"))
```

```
## Random Forest Training Error: 0
##  Random Forest Test Error: 0.217391304347826
```

```r
rf_test_confusion <- table(Predicted = rf_test_pred, Actual = testData$diabetes)

cat("\nTest Confusion Matrix:\n")
```

```
##
## Test Confusion Matrix:
```

```r
print(rf_test_confusion)
```

```
##          Actual
## Predicted neg pos
##       neg 133  23
##       pos  32  65
```

```r
cat(paste("Training Error:", rf_train_error, "\n",
          "Test Error:", rf_test_error, "\n"))
```

```
## Training Error: 0
##  Test Error: 0.217391304347826
```

The classification tree and random forest had the same test error of 0.217, while the bagged trees model showed the highest test error at 0.237. Both ensemble methods (bagged trees and random forest) achieved perfect or near-perfect performance on the training data, but their test error rates did not outperform the simpler classification tree. This suggests that although the ensemble methods reduce variance by averaging multiple trees, they do not necessarily improve generalization on this specific dataset.

Neural Network Modeling for Diabetes Prediction

Now, as last model, we look at Neural Networks. Here we perform a neural network where the number of neurons in the hidden layer is varied to study its effect on both training and test error rates. The sequence of neurons ranges from 1 to 100, and for each neuron count, a neural network is trained on scaled training data (`trainData_scaled`). The model is fitted with a maximum of 500 iterations, using logistic activation since it is a classification task. For each configuration, predictions are made on both the training and test sets, and classification errors are calculated. Once all models are trained, the optimal number of neurons is identified by selecting the configuration that minimizes the test error.

```
library(nnet)
library(caret)

set.seed(2024)
neurons <- seq(1, 100, by = 5)
train_errors <- c()
test_errors <- c()

# Loop
for (n in neurons) {
  # Fit
  nn_model <- nnet(diabetes ~ ., data = trainData_scaled,
                   size = n, linout = FALSE, maxit = 500, trace = FALSE)
```

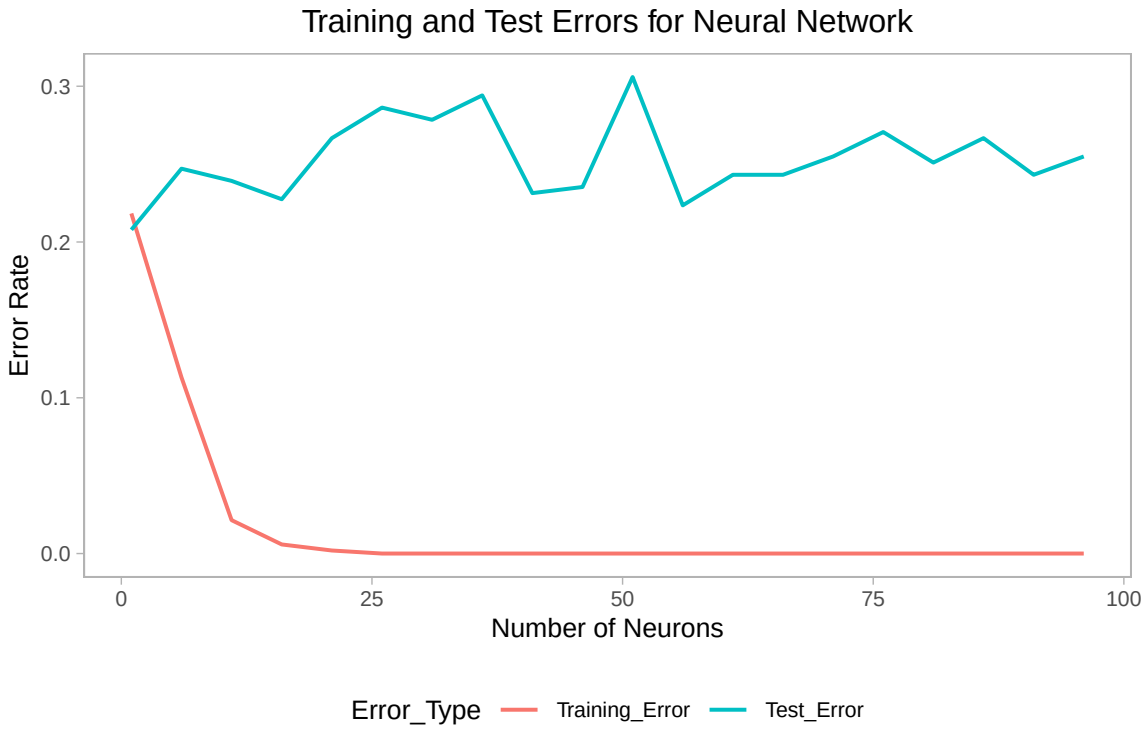
```
  # Train Pred
  train_pred <- predict(nn_model, trainData_scaled)
  train_pred_class <- ifelse(train_pred > 0.5, 1, 0)
  train_error <- mean(train_pred_class != as.numeric(levels(
    trainData_scaled$diabetes))[trainData_scaled$diabetes])
  train_errors <- c(train_errors, train_error)

  # Test Pred
  test_pred <- predict(nn_model, testData_scaled)
  test_pred_class <- ifelse(test_pred > 0.5, 1, 0)
  test_error <- mean(test_pred_class != as.numeric(levels(
    testData_scaled$diabetes))[testData_scaled$diabetes])
  test_errors <- c(test_errors, test_error)
}
```

```
optimal_neurons <- neurons[which.min(test_errors)]
cat("Optimal number of neurons:", optimal_neurons, "\n")
```

```
## Optimal number of neurons: 1
```

The graph illustrates the training and test error rates across different neuron configurations. The training error decreases quickly as the number of neurons increases, reaching zero around 25 neurons and staying flat at zero afterward. The model is able to perfectly fit the training data beyond a certain complexity, a sign of potential overfitting. On the other hand, the test error initially decreases, suggesting that the model is generalizing well to the unseen data. However, after around 25 neurons, the test error begins to fluctuate and increase.

```
library(ggplot2)
error_results_long <- reshape2::melt(error_results,
                                     id.vars = "Neurons",
                                     variable.name = "Error_Type",
                                     value.name = "Error")

ggplot(error_results_long, aes(x = Neurons, y = Error,
                               color = Error_Type)) +
  geom_line(size = 0.7) +
  labs(title = "Training and Test Errors for Neural Network",
       x = "Number of Neurons",
       y = "Error Rate") +
  theme_light() +
  theme(legend.position = "bottom",
        panel.grid = element_blank(),
        plot.title = element_text(hjust = 0.5),
        text = element_text(size = 10))
```



The behavior shown in the graph is a classic case of overfitting. The rapid decline in training error and the rising fluctuations in test error suggest that adding more neurons beyond a certain point results in a model that is too finely tuned to the training data, leading to poor generalization.

The results from the confusion matrix, based on a model trained with 1 neuron, reveal that the model has a training error of 0.218 and a test error of 0.207. The confusion matrix shows a balanced accuracy of 0.7922 and a Kappa statistic of 0.5366 (moderate agreement between predicted and true labels). The model demonstrates high sensitivity (85.54%) but lower specificity (67.42%), basically it performs better at identifying non-diabetic cases than diabetic ones.

```
cat(paste("Training Error for 1 neurons:", train_error, "\n",
          "Test Error for 1 neurons:", test_error, "\n"))
```

```
## Training Error for 1 neurons: 0.218323586744639
##  Test Error for 1 neurons: 0.207843137254902
```

```
confusion <- confusionMatrix(as.factor(test_pred_class), testData$diabetes)
print(confusion)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 142  29
##          1  24  60
##
##                Accuracy : 0.7922
##                  95% CI : (0.7371, 0.8403)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 6.053e-07
##
```

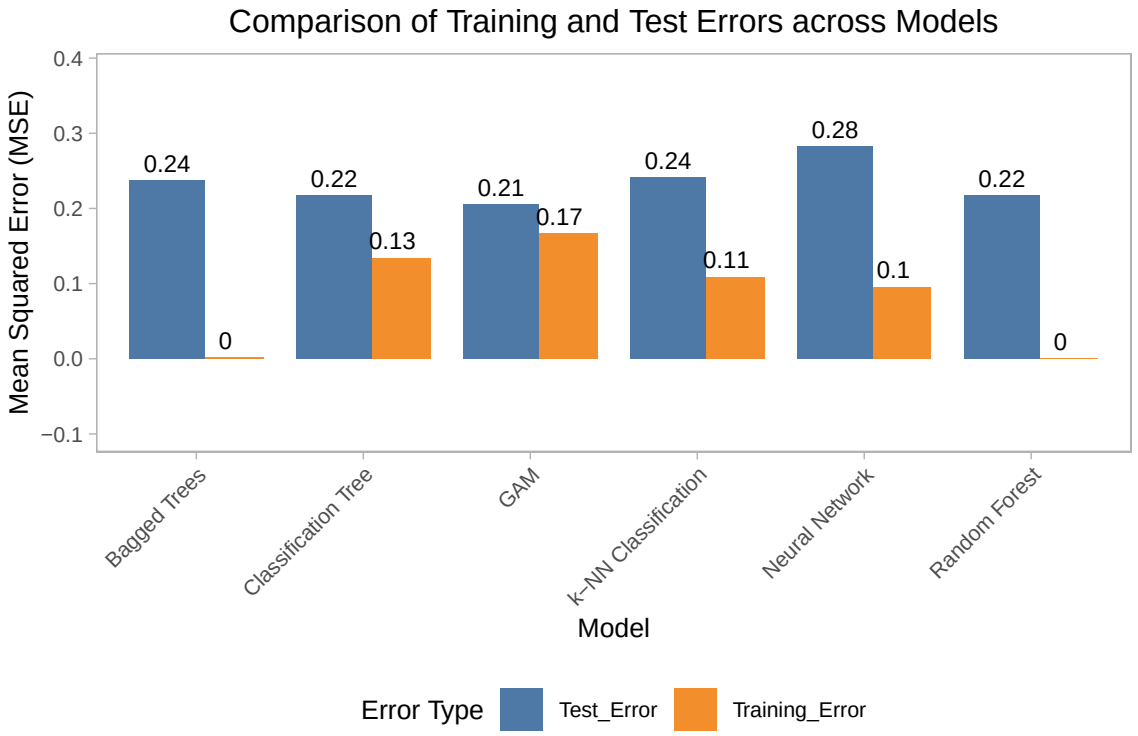
```
##                   Kappa : 0.5366
##
##  Mcnemar's Test P-Value : 0.5827
##
##             Sensitivity : 0.8554
##             Specificity : 0.6742
##          Pos Pred Value : 0.8304
##          Neg Pred Value : 0.7143
##              Prevalence : 0.6510
##          Detection Rate : 0.5569
##    Detection Prevalence : 0.6706
##       Balanced Accuracy : 0.7648
##
##        'Positive' Class : 0
##
```

This case serves as a practical demonstration of the Universal Approximation Theorem, which asserts that a neural network with a sufficient number of neurons can approximate any continuous function. As seen in the experiment, the training error rapidly decreases as the number of neurons increases, reaching zero around 25 neurons, indicating that the network is able to perfectly capture the patterns in the training data. This supports the theorem's claim that a neural network, with enough capacity, can approximate complex decision boundaries. However, as the number of neurons continues to increase beyond this point, the test error begins to fluctuate and rise, which suggests overfitting. While the Universal Approximation Theorem guarantees that a network can model any function, it does not guarantee generalization. In this case, the network starts fitting not only the true patterns but also the noise in the training data, leading to worse performance on unseen data. Interestingly, the model with just one neuron provides the best test error, showing that simpler models often generalize better, as they avoid the risk of overfitting that comes with more complex networks.

Comparing Classification Methods for Diabetes Analysis

In this final section, we present a comparative analysis of the various classification models that we used for the problem of diabetes prediction, focusing on model performance in terms of training and test errors.

The first graph displays the Mean Squared Error for training and test datasets across the six models: Bagged Trees, Classification Tree, Generalized Additive Model (GAM), k-Nearest Neighbors (k-NN) Classification, Neural Network, and Random Forest. A clear pattern emerges: Bagged Trees and Random Forest models achieve zero training error, indicating that these ensemble methods can perfectly fit the training data. However, the test errors for these models (0.24 and 0.22, respectively) are notably higher than their training counterparts, suggesting overfitting, where the model captures noise in the training data, which does not generalize well to new data. The GAM and k-NN models show a smaller gap between training and test errors, with the k-NN model, in particular, achieving the lowest test error (0.11), indicating it might be the most effective model in terms of generalization for this task. The Neural Network model also shows a discrepancy, with a higher test error (0.28) relative to its training error (0.1), hinting at potential overfitting.



In conclusion, the analysis of this data reveals that the k-NN model provides the best generalization, as evidenced by the lowest test error and highest test accuracy, suggesting it may be the most suitable model for this diabetes classification task. In contrast, Bagged Trees, Random Forest, and Neural Network, while highly accurate on training data, display significant drops in test performance, indicating overfitting and potential challenges in generalization. These findings align with the expectations for ensemble and neural network methods, where high model complexity can lead to perfect training results but poorer test performance if not carefully regularized.