OJ Purchase Prediction Using XGBoost

Nikhil Prema Chandra Rao

2024-11-10

Introduction

In this project, we create a prediction model using the OJ dataset from the ISLR2 library. The aim is to guess if a customer will buy Citrus Hill (CH) or Minute Maid (MM) orange juice by looking at things like the price, information about the store, and other important details.

We use XGBoost, a strong and fast tool, for making classifications. Gradient boosting takes the results from several simple models to make more accurate predictions. XGBoost is especially good at managing big data and improving how well predictions are made.

Data Pre-processing

First, we will encode the `Purchase` variable as a binary target, with `CH` as 1 and `MM` as 0, and split the data set into training and test sets (50:50 ratio).

```
# Set seed for reproducibility
set.seed(555)

# Encode Purchase as binary: 1 if "CH", 0 if "MM"
OJ$Purchase <- as.numeric(OJ$Purchase == "CH")

str(OJ)
```

```
## 'data.frame':    1070 obs. of  18 variables:
##  $ Purchase      : num  1 1 1 0 1 1 1 1 1 1 ...
##  $ WeekofPurchase: num  237 239 245 227 228 230 232 234 235 238 ...
##  $ StoreID       : num  1 1 1 1 7 7 7 7 7 7 ...
##  $ PriceCH       : num  1.75 1.75 1.86 1.69 1.69 1.69 1.69 1.75 1.75 1.75 ...
##  $ PriceMM       : num  1.99 1.99 2.09 1.69 1.69 1.99 1.99 1.99 1.99 1.99 ...
##  $ DiscCH        : num  0 0 0.17 0 0 0 0 0 0 0 ...
##  $ DiscMM        : num  0 0.3 0 0 0 0 0.4 0.4 0.4 0.4 ...
##  $ SpecialCH     : num  0 0 0 0 0 0 1 1 0 0 ...
##  $ SpecialMM     : num  0 1 0 0 0 1 1 0 0 0 ...
##  $ LoyalCH       : num  0.5 0.6 0.68 0.4 0.957 ...
##  $ SalePriceMM   : num  1.99 1.69 2.09 1.69 1.69 1.99 1.59 1.59 1.59 1.59 ...
##  $ SalePriceCH   : num  1.75 1.75 1.69 1.69 1.69 1.69 1.69 1.75 1.75 1.75 ...
##  $ PriceDiff     : num  0.24 -0.06 0.4 0 0 0.3 -0.1 -0.16 -0.16 -0.16 ...
##  $ Store7        : Factor w/ 2 levels "No","Yes": 1 1 1 1 2 2 2 2 2 2 ...
##  $ PctDiscMM     : num  0 0.151 0 0 0 ...
##  $ PctDiscCH     : num  0 0 0.0914 0 0 ...
```

```
##  $ ListPriceDiff : num  0.24 0.24 0.23 0 0 0.3 0.3 0.24 0.24 0.24 ...
##  $ STORE         : num  1 1 1 1 0 0 0 0 0 0 ...
```

```
?OJ
sum(is.na(OJ))
```

```
## [1] 0
```

Key Variables in the Dataset

- **Purchase**: The binary target variable (1 for CH, 0 for MM), which we encoded earlier.
- **WeekofPurchase**: The week number when the purchase was made.
- **StoreID**: The ID of the store where the purchase took place.
- **PriceCH**: Price of Citrus Hill (CH) in that week.
- **PriceMM**: Price of Minute Maid (MM) in that week.
- **DiscCH**: Discount on Citrus Hill during that week.
- **DiscMM**: Discount on Minute Maid during that week.
- **SpecialCH**: Whether Citrus Hill was on special promotion.
- **SpecialMM**: Whether Minute Maid was on special promotion.
- **LoyalCH**: A loyalty score for customers buying Citrus Hill.
- **SalePriceCH**: Final sale price for Citrus Hill after discounts.
- **SalePriceMM**: Final sale price for Minute Maid after discounts.
- **PriceDiff**: The difference between the sale prices of Citrus Hill and Minute Maid.
- **Store7**: A factor indicating whether the purchase occurred at store 7 (Yes/No).
- **PctDiscMM**: Percentage discount on Minute Maid.
- **PctDiscCH**: Percentage discount on Citrus Hill.
- **ListPriceDiff**: Difference in list prices between Citrus Hill and Minute Maid (before discounts).
- **STORE**: Additional store information.

Encoding the Target Variable

The `Purchase` column originally contains categorical values: - `"CH"` for Citrus Hill and - `"MM"` for Minute Maid.

We convert this column into a numeric binary variable: - `1` for Citrus Hill ("CH") - `0` for Minute Maid ("MM")

This is achieved by creating a logical vector with `== "CH"`, which is then converted into `1` or `0` using the `as.numeric()` function.

The data set has 1,070 observations with 18 variables, and the Purchase variable is now binary (1 for Citrus Hill, 0 for Minute Maid). All variables are numeric or factor types, and there are no missing values. This sets the dataset up for predictive modeling using machine learning techniques like XGBoost.

Prepare the Dataset:

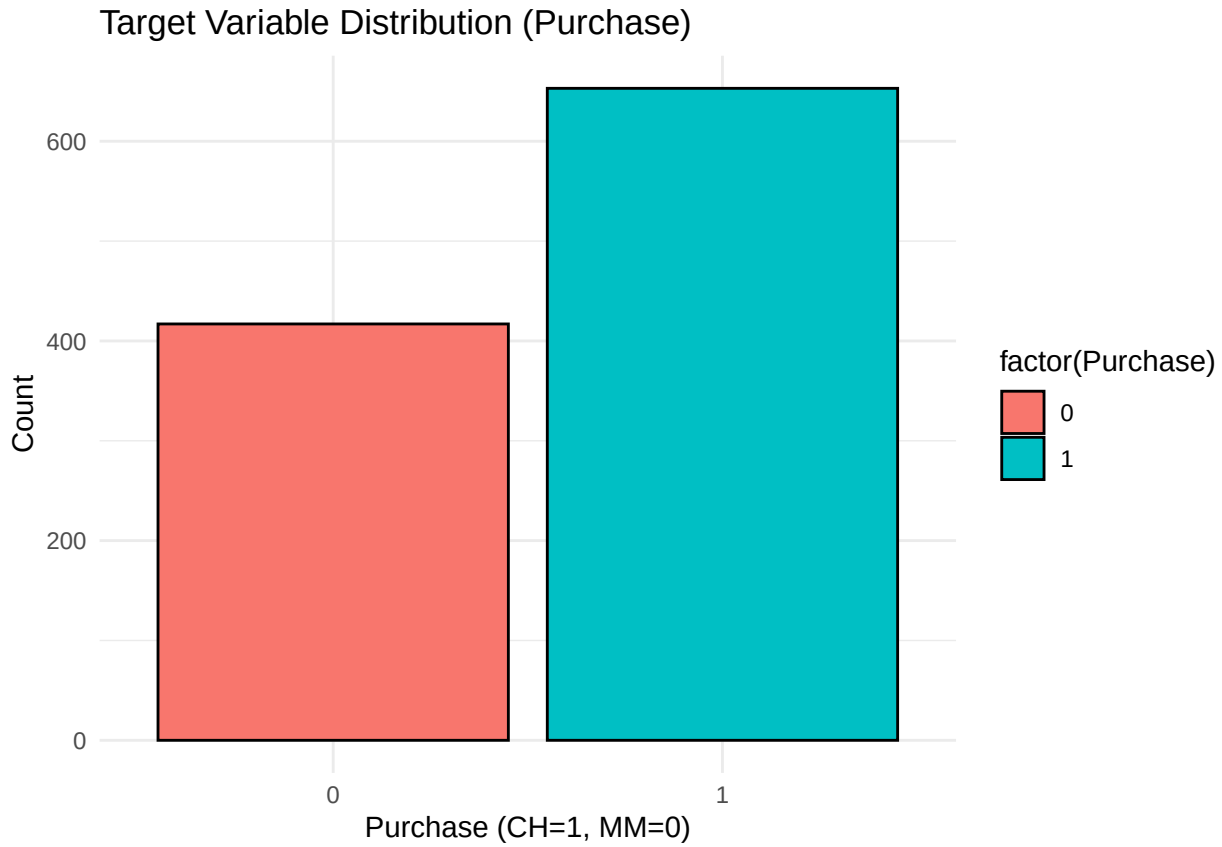
```
# Split the dataset into 50:50 train-test sets
trainIndex <- createDataPartition(OJ$Purchase, p = 0.5, list = FALSE)
train_data <- OJ[trainIndex, ]
test_data <- OJ[-trainIndex, ]
```

This code divides the `OJ` dataset into two parts: training and testing. `trainIndex` randomly picks 50% of the data based on the `Purchase` variable. Then, `train_data` is made using the rows from `trainIndex` (for training), and `test_data` is made by using the rows that are not in `trainIndex` (for testing).

Visualize Target Variable Distribution

Target Variable Distribution (Purchase)

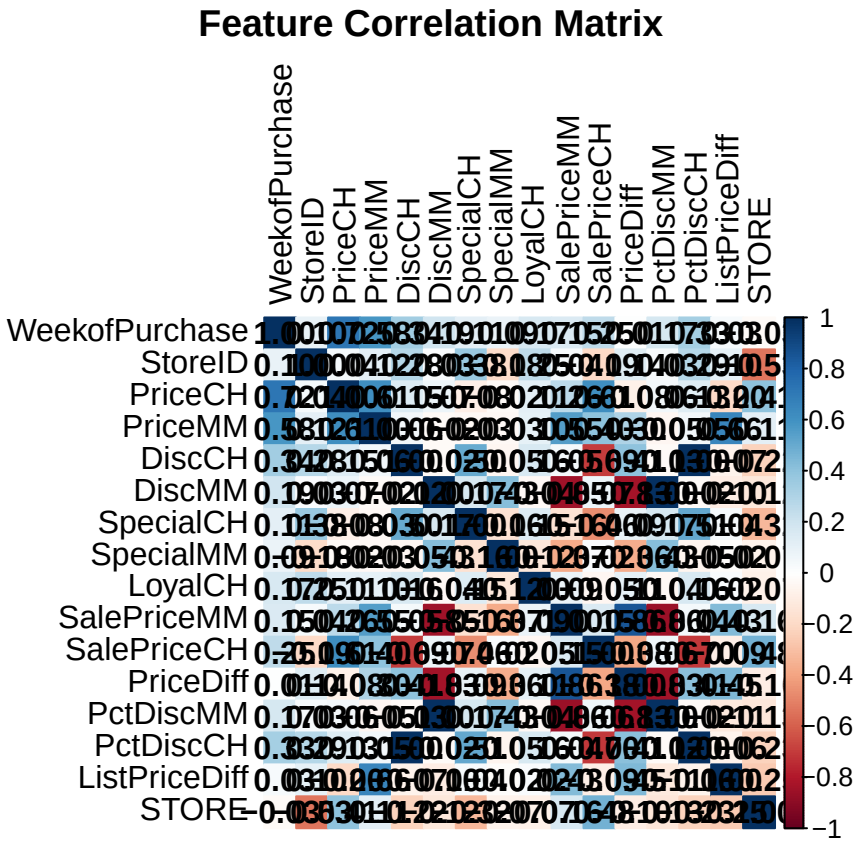
```
ggplot(OJ, aes(x = factor(Purchase))) +
  geom_bar(aes(fill = factor(Purchase)), color = "black") +
  labs(title = "Target Variable Distribution (Purchase)", x = "Purchase (CH=1, MM=0)", y = "Count") +
  theme_minimal()
```



The picture is a bar graph showing how a variable called "Purchase" is spread out. This variable has two options: 0 and 1. This variable shows a buying decision, where 0 means choosing "MM" and 1 means choosing "CH." Each bar shows how many times each choice was made, with the red bar for "0" (MM choice) and the turquoise bar for "1" (CH choice). The height of each bar shows how many times something happened in each category. The bar for "1" (CH) is taller than the bar for "0" (MM), which means more people chose "CH" than "MM" when making their purchase decisions in this data. This distribution shows how the two groups in the target variable compare to each other. This is important for checking if there is any bias or imbalance in the dataset, especially when dealing with a classification problem.

Feature Correlation Matrix

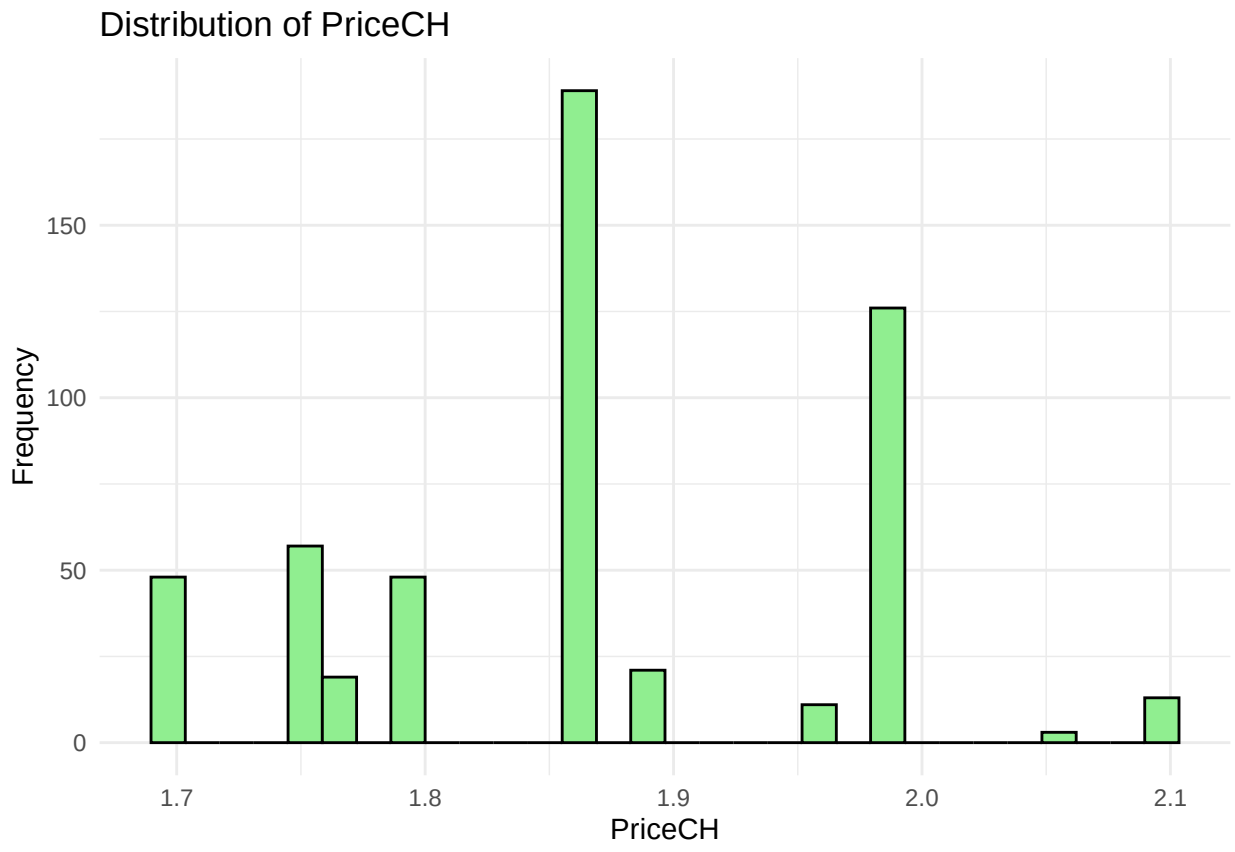
```
numeric_features <- train_data %>%
  select(where(is.numeric)) %>%
  select(-Purchase)  # Exclude the target variable for correlation

cor_matrix <- cor(numeric_features, use = "complete.obs")
corrplot(cor_matrix, method = "color", addCoef.col = "black", tl.col = "black",
         title = "Feature Correlation Matrix", mar = c(0, 0, 2, 0))
```



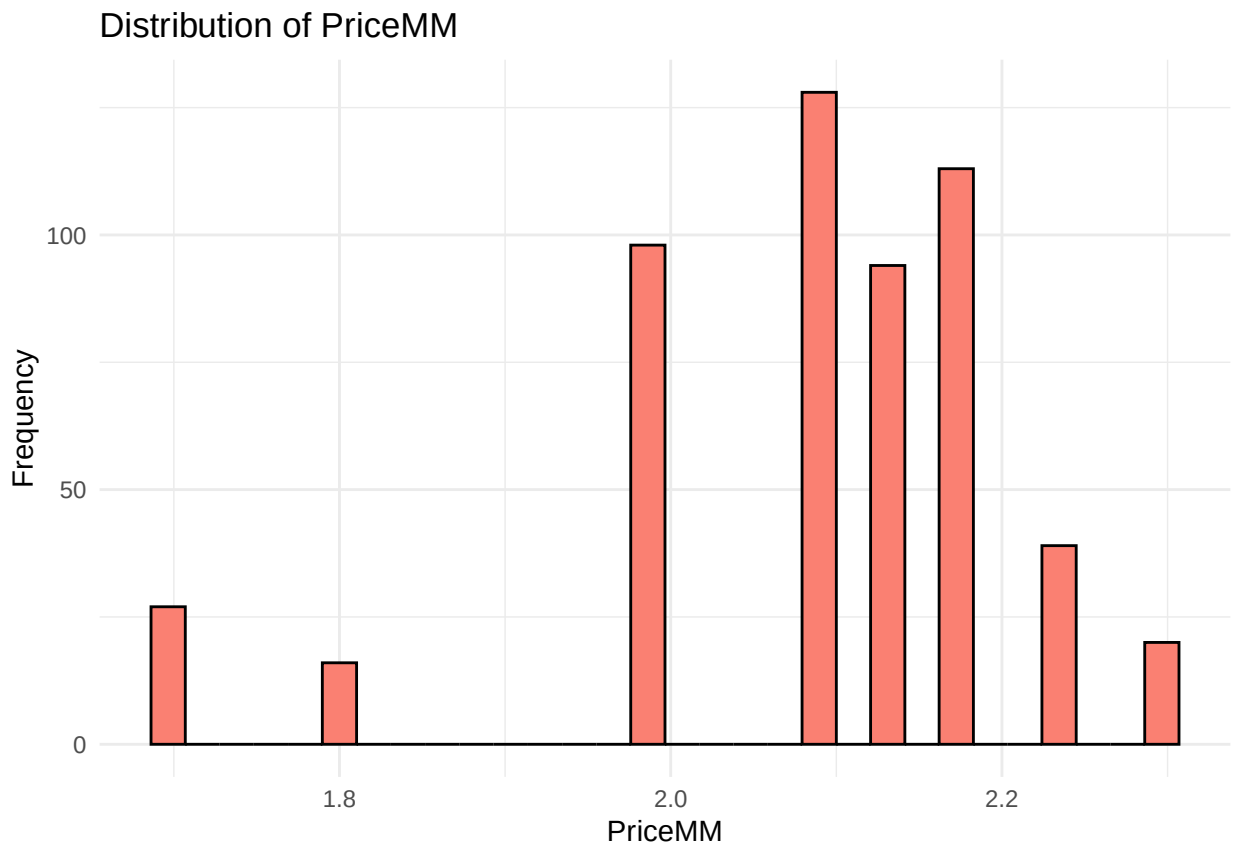
This is a heatmap showing how different parts of a dataset are related to each other. The matrix is a square chart that shows how two features are related to each other. Each box in the chart has a number between -1 and 1 that represents this relationship. A correlation coefficient of +1 means there is a perfect positive relationship between two things, while -1 means there is a perfect negative relationship. In this chart, darker blue colors show strong positive connections, meaning that when one thing goes up, the other usually goes up too. On the other hand, darker red colors show a strong negative connection, meaning that when one thing goes up, the other goes down. For example, "PriceCH" and "PriceMM" have a strong positive relationship, shown by a dark blue cell. On the other hand, "DiscCH" and "SpecialMM" have a negative relationship, indicated by a red cell. The diagonal of the matrix shows how each feature relates to itself, which always has a correlation of 1 (the darkest blue). The color bar on the right shows the range of correlation values from -1 to +1. It helps you understand what each cell's color means. This chart helps identify features that are closely related to each other, which is important for choosing the right features or preparing the data.

```r
# Convert categorical variables to numeric using model.matrix
train_x <- model.matrix(Purchase ~ . - 1, data = train_data)  # Exclude intercept
test_x <- model.matrix(Purchase ~ . - 1, data = test_data)

# Extract target variables
train_y <- train_data$Purchase
test_y <- test_data$Purchase

# Convert datasets to DMatrix format
dtrain <- xgb.DMatrix(data = train_x, label = train_y)
dtest <- xgb.DMatrix(data = test_x, label = test_y)
```

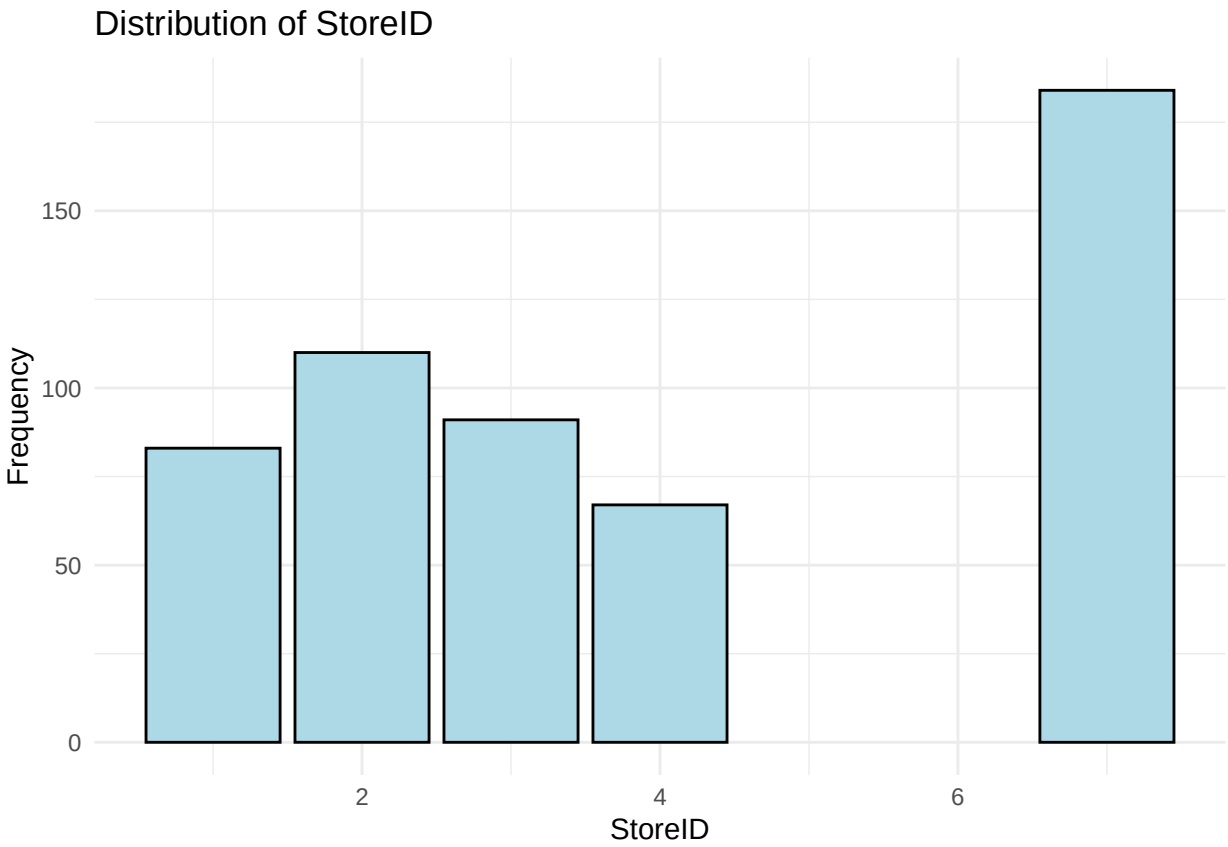
```r
# Plot feature distributions (e.g., PriceCH, PriceMM, and StoreID)
ggplot(train_data, aes(x = PriceCH)) +
  geom_histogram(bins = 30, fill = "lightgreen", color = "black") +
  labs(title = "Distribution of PriceCH", x = "PriceCH", y = "Frequency") +
  theme_minimal()
```




```r
ggplot(train_data, aes(x = PriceMM)) +
  geom_histogram(bins = 30, fill = "salmon", color = "black") +
  labs(title = "Distribution of PriceMM", x = "PriceMM", y = "Frequency") +
  theme_minimal()
```




```
ggplot(train_data, aes(x = StoreID)) +
  geom_bar(fill = "lightblue", color = "black") +
  labs(title = "Distribution of StoreID", x = "StoreID", y = "Frequency") +
  theme_minimal()
```



1. This bar chart shows how often different price ranges appear in the "PriceCH" data. Most of the values are grouped around certain prices, with significant highs at prices like 1. 7, 18, and 20, along with some smaller highs at other prices. This pattern shows that some price levels for "PriceCH" happen more often. This could be because of the pricing strategies, what people want, or other factors affecting the prices.

2. The bar chart for "PriceMM" shows how often different price ranges occur. The data is grouped around certain price levels, with the most common prices being between 2. 0 and 22 This distribution shows that most data points are within this range, suggesting a consistent price area for "PriceMM." This could be due to similar reasons as those affecting "PriceCH."

3. StoreID Distribution: This bar chart shows how many times each "StoreID" appears, indicating the number of observations for each store. The chart shows that there are big differences in how often different stores are visited. StoreID 6 has the most visits, then StoreID 2, and the others have fewer visits. This might mean that some stores have more data because they sell more or have more customers. It suggests that the data is not evenly spread out among the stores.

Training the Model

```
# Set base parameters for XGBoost
params <- list(
  objective = "binary:logistic",  # Binary classification
  eval_metric = "error",          # Error rate
  max_depth = 6,                  # Tree depth
```

```
  eta = 0.3,                          # Learning rate
  nthread = 2                         # Number of threads
)

# Train the XGBoost model
xgb_model <- xgb.train(
  params = params,
  data = dtrain,
  nrounds = 100,  # Number of boosting rounds
  watchlist = list(train = dtrain, test = dtest),
  early_stopping_rounds = 10,  # Stop early if no improvement
  verbose = 1
)
```

```
## [1]  train-error:0.084112    test-error:0.216822
## Multiple eval metrics are present. Will use test_error for early stopping.
## Will train until test_error hasn't improved in 10 rounds.
##
## [2]  train-error:0.085981    test-error:0.214953
## [3]  train-error:0.072897    test-error:0.211215
## [4]  train-error:0.069159    test-error:0.213084
## [5]  train-error:0.065421    test-error:0.211215
## [6]  train-error:0.065421    test-error:0.209346
## [7]  train-error:0.063551    test-error:0.205607
## [8]  train-error:0.065421    test-error:0.214953
## [9]  train-error:0.065421    test-error:0.209346
## [10] train-error:0.065421    test-error:0.209346
## [11] train-error:0.061682    test-error:0.209346
## [12] train-error:0.059813    test-error:0.211215
## [13] train-error:0.057944    test-error:0.209346
## [14] train-error:0.056075    test-error:0.211215
## [15] train-error:0.048598    test-error:0.209346
## [16] train-error:0.044860    test-error:0.216822
## [17] train-error:0.039252    test-error:0.222430
## Stopping. Best iteration:
## [7]  train-error:0.063551    test-error:0.205607
```

```
# Make predictions on the test set
pred_prob <- predict(xgb_model, dtest)
pred_class <- ifelse(pred_prob > 0.5, 1, 0)
```

The code sets up the parameters for training the XGBoost model, including:

- `objective = "binary:logistic"`: This specifies the problem is a binary classification (predicting 0 or 1).
- `eval_metric = "error"`: The evaluation metric used is the error rate (percentage of incorrect predictions).
- `max_depth = 6`: The maximum depth of the decision trees in the model, controlling model complexity.
- `eta = 0.3`: The learning rate, which controls how much the model is adjusted in each boosting round.
- `nthread = 2`: Specifies the number of CPU threads to be used for training.

The `xgb.train` function is called to train the XGBoost model using the parameters defined earlier. It uses the training data (`dtrain`), performs 100 boosting rounds, and includes a watchlist for both train-

ing and test datasets. The model will stop early if the test error doesn't improve over 10 rounds (via `early_stopping_rounds`).

Making Predictions

After the model is trained, the `predict` function generates probability predictions on the test set (`dtest`). The `ifelse` function then converts these probabilities into binary predictions (1 for "CH" and 0 for "MM") by setting a threshold of 0.5.

Output

The output logs show the progression of training over each boosting round, with the evaluation of both **train-error** (error rate on training data) and - **test-error** (error rate on testing data).

- **Train-error** represents the error rate on the training dataset, while **test-error** represents the error rate on the test dataset. For example, in the first round: "[1] train-error:0.084112 test-error:0.21682"
- The model has a **train error** of 8.41% and a **test error** of 21.68% at the end of the first boosting round.

The logs continue for each boosting round, showing how the errors change with each round of training. Here's a breakdown of the important parts:

Early Stopping

- The message "Multiple eval metrics are present. Will use test_error for early stopping" indicates that the model will use the test error to determine when to stop training early.
- The training stops at round 17 due to **early stopping**, as the test error didn't improve for 10 rounds in a row. The best model is found at round 7 (since that's where the lowest test error occurred).

Test Error

- As training progresses, the **test-error** initially decreases, reaching a minimum of **20.56%** at round 7.
- After round 7, the test error begins to increase again (e.g., round 8 shows a test error of 21.50%), signaling that further training may lead to overfitting.

Best Iteration

- The model training halts at round 17 due to early stopping. The **best iteration** is round 7, where the **test-error** was the lowest at **20.56%**.

This log demonstrates the model's learning process, the effect of early stopping, and the importance of evaluating both training and test errors during model training.

```
# Evaluate the model with confusion matrix for initial model
conf_matrix_base <- table(Prediction = pred_class, Reference = test_y)
conf_matrix_base
```

```
##           Reference
## Prediction   0   1
##          0 164  62
##          1  48 261
```

```r
accuracy_base <- sum(diag(conf_matrix_base)) / sum(conf_matrix_base)
accuracy_base
```

```
## [1] 0.7943925
```

Model Performance Evaluation

In this document, we calculate and evaluate the accuracy of a model based on its confusion matrix.

Confusion Matrix The confusion matrix provides a summary of the predictions made by the model. The entries are defined as follows:

- **True Negatives (TN)**: 164 (The model correctly predicted 0 as 0)
- **False Positives (FP)**: 62 (The model incorrectly predicted 0 as 1)
- **False Negatives (FN)**: 48 (The model incorrectly predicted 1 as 0)
- **True Positives (TP)**: 261 (The model correctly predicted 1 as 1)

Formula for Accuracy The accuracy is calculated as the proportion of correct predictions (TP + TN) to the total number of predictions. The formula for accuracy is:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where: - **TP** = True Positives - **TN** = True Negatives - **FP** = False Positives - **FN** = False Negatives

Calculation of Accuracy We calculate accuracy using the formula above. The accuracy is computed as follows:

Confusion matrix for the model conf_matrix_base <- matrix(c(164, 62, 48, 261), nrow = 2)

Calculate accuracy The function `diag(conf_matrix_base)` extracts the diagonal elements (True Positives (TP) + True Negatives (TN)), representing the correctly classified instances:

$$TP + TN = 164 + 261 = 425$$

Next, `sum(conf_matrix_base)` gives the total number of observations:

$$TP + TN + FP + FN = 164 + 62 + 48 + 261 = 535$$

Thus, the accuracy is calculated as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{425}{535} \approx 0.7943925$$

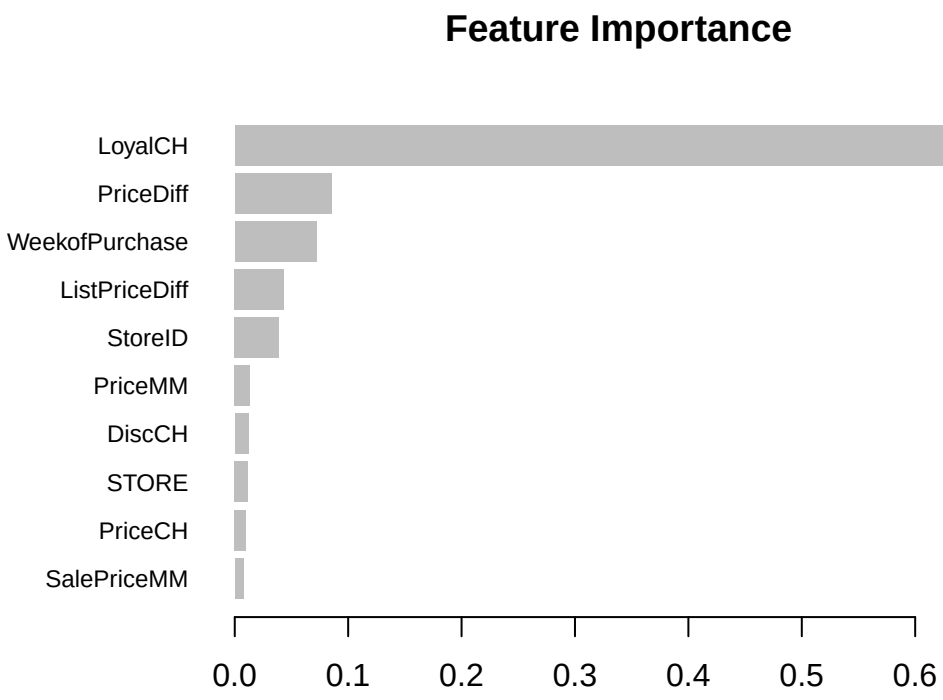
This means that the model correctly predicted the class labels 79.44% of the time.

Summary

The confusion matrix provides insight into the number of correct and incorrect predictions made by the model for both classes (0 and 1). The accuracy of the model is approximately 79.44%, indicating that the model correctly classified the target variable about 79% of the time.

Feature Importance plot

```
importance_matrix <- xgb.importance(feature_names = colnames(train_x), model = xgb_model)
xgb.plot.importance(importance_matrix, main = "Feature Importance", top_n = 10)
```



This bar chart shows how important different features are in a prediction model. The x-axis shows how important each feature is, with higher numbers meaning it's more important. The most important part is "LoyalCH," which is much more important than anything else. This means that how loyal customers are to CH (maybe a brand or shop) really affects what the model predicts. Some other important features are "PriceDiff," "WeekofPurchase," and "ListPriceDiff." These are somewhat important for the model, but not as much as "LoyalCH." The other features, like "StoreID," "PriceMM," "DiscCH," "STORE," "PriceCH," and "SalePriceMM," are not very important and have little effect on what the model predicts. The story shows that "LoyalCH" is the best at predicting things compared to the other options.

Plot the ROC curve

```r
test_preds_prob <- predict(xgb_model, dtest)
roc_curve <- roc(test_y, test_preds_prob)
```

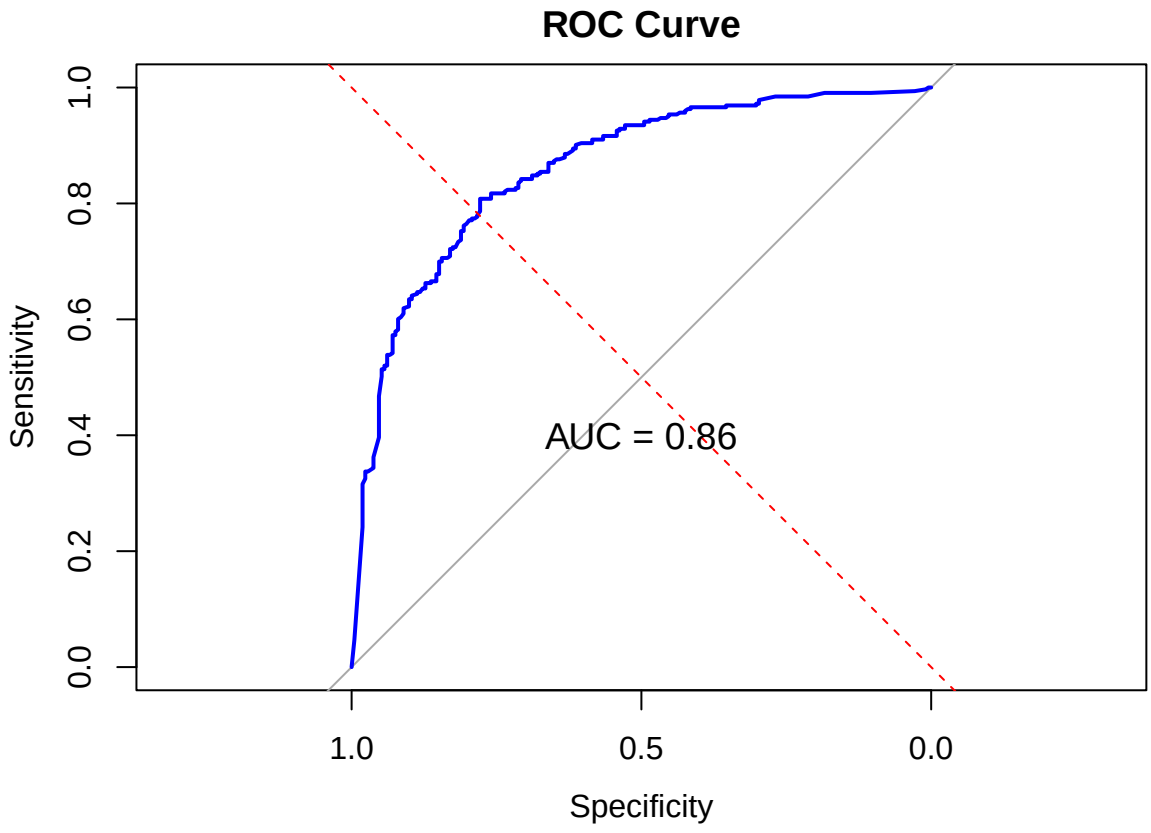
```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```r
roc_curve
```

```
##
## Call:
## roc.default(response = test_y, predictor = test_preds_prob)
##
## Data: test_preds_prob in 212 controls (test_y 0) < 323 cases (test_y 1).
## Area under the curve: 0.8598
```

```r
plot(roc_curve, col = "blue", main = "ROC Curve", lwd = 2)
abline(a = 0, b = 1, col = "red", lty = 2)
text(0.5, 0.4, paste("AUC =", round(roc_curve$auc, 3)), cex = 1.2, col = "black")
```



This image shows an ROC curve, which helps us see how well a model works when it has to classify things into two groups. It shows the true positive rate (how well it identifies the correct positive cases) on the vertical side and the false positive rate (how often it wrongly identifies negatives as positives) on the horizontal side.

The blue line shows how well the model works at different settings, and the grey diagonal line shows how a random guess would perform (AUC = 0. 5) The area under the curve (AUC) is 0. 86, showing that the model does a good job of telling the difference between the two groups (212 controls and 323 cases). An AUC of 0. 86 shows that the model is good at telling the difference between positive and negative cases. A higher AUC means the model is even better at classifying them. The red dashed line shows the point of no discrimination. The model's ROC curve is above this line, which means it has very good accuracy in making predictions.

Cross-Validation for Hyper-parameter Tuning

```
# Perform cross-validation to find the best parameters
xgb_cv <- xgb.cv(
  params = params,
  data = dtrain,
  nrounds = 1000,
  nfold = 5,  # 5-fold cross-validation
  early_stopping_rounds = 10,
  verbose = 1
)
```

```
## [1]  train-error:0.085514+0.008174   test-error:0.173832+0.029317
## Multiple eval metrics are present. Will use test_error for early stopping.
## Will train until test_error hasn't improved in 10 rounds.
##
## [2]  train-error:0.074766+0.009690   test-error:0.153271+0.023346
## [3]  train-error:0.076636+0.008667   test-error:0.166355+0.021637
## [4]  train-error:0.070561+0.007148   test-error:0.157009+0.030370
## [5]  train-error:0.067290+0.004531   test-error:0.153271+0.030485
## [6]  train-error:0.062150+0.003796   test-error:0.153271+0.033749
## [7]  train-error:0.060280+0.005409   test-error:0.149533+0.030713
## [8]  train-error:0.054673+0.006373   test-error:0.151402+0.034160
## [9]  train-error:0.050000+0.004578   test-error:0.153271+0.034768
## [10] train-error:0.047664+0.006707   test-error:0.157009+0.034668
## [11] train-error:0.045327+0.007025   test-error:0.158879+0.033955
## [12] train-error:0.044860+0.007593   test-error:0.158879+0.035954
## [13] train-error:0.042523+0.006674   test-error:0.164486+0.033227
## [14] train-error:0.042523+0.009037   test-error:0.170093+0.027343
## [15] train-error:0.042991+0.008692   test-error:0.170093+0.025355
## [16] train-error:0.039252+0.008667   test-error:0.166355+0.029789
## [17] train-error:0.035981+0.005836   test-error:0.158879+0.027724
## Stopping. Best iteration:
## [7]  train-error:0.060280+0.005409   test-error:0.149533+0.030713
```

```
# Get the best number of rounds from cross-validation
best_nrounds <- xgb_cv$best_iteration
```

The output shown here is from the cross-validation process during the training of the XGBoost model. The `train-error` and `test-error` represent the error rates on the training and test sets, respectively, over each iteration. These errors are displayed along with their standard deviations, indicating the variability in the error across the 5 folds used in the 5-fold cross-validation.

- **train-error**: This is the error rate on the training data for each boosting round.

- **test-error**: This is the error rate on the test data for each boosting round.
- After each error rate represents the standard deviation of the error across the folds.

Additionally, the output indicates that multiple evaluation metrics are present, but the model is using `test-error` for early stopping. The model will stop training once the `test-error` has not improved for a certain number of rounds, specified as 10 (`early_stopping_rounds = 10`).

Output

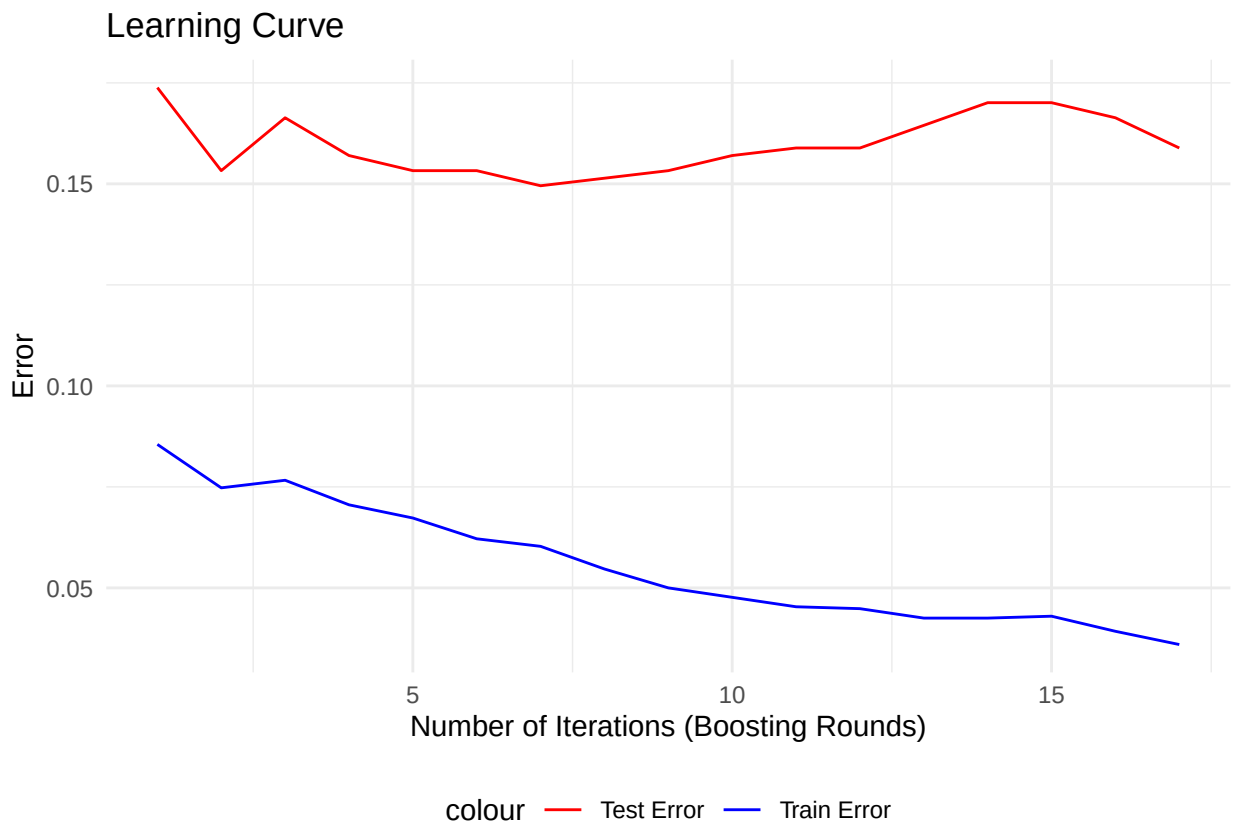
1. **Initial Rounds (1 to 6)**: During the early iterations, both the training and test errors decrease, with the training error starting at around 0.085 and decreasing to 0.062. Meanwhile, the test error also drops from 0.173 to 0.153, showing an improvement in the model's generalization ability.
2. **Intermediate Rounds (7 to 16)**: After the first few rounds, the training error continues to decrease slowly, reaching 0.039 at iteration 16, while the test error fluctuates slightly. The test error, however, reaches a minimum of 0.1495 at iteration 7 and shows some increase in subsequent rounds.
3. **Early Stopping**: Since the test error at round 7 (0.1495) is the lowest observed, and no further improvement occurs for the next few rounds (iterations 8-17), early stopping is triggered after round 17. The best model is selected from iteration 7, where the test error was the lowest.

Best Iteration The best iteration is found to be **round 7**, where the model achieved the lowest test error of **0.149533**, with a standard deviation of **0.030713**.

In summary, the cross-validation process helped identify that training beyond 7 rounds did not improve the test error, and thus early stopping was employed to prevent overfitting.

Learning curve plot using cross-validation error

```
cv_error <- xgb_cv$evaluation_log
ggplot(cv_error, aes(x = iter)) +
  geom_line(aes(y = train_error_mean, color = "Train Error")) +
  geom_line(aes(y = test_error_mean, color = "Test Error")) +
  labs(title = "Learning Curve", x = "Number of Iterations (Boosting Rounds)", y = "Error") +
  theme_minimal() +
  scale_color_manual(values = c("Train Error" = "blue", "Test Error" = "red")) +
  theme(legend.position = "bottom")
```



This image shows a graph that tracks how well a boosting model learns over time. It displays the training and test errors based on the number of boosting rounds (or tries) made in the model. The bottom line shows how many times something happens, and the side line shows how often there is a mistake. The blue line shows the training error, which goes down steadily as the number of tries increases. This means the model is learning well from the training data. The red line shows the test error. At first, it goes up and down a lot, but then it stays mostly the same, with just a small drop as time goes on. The difference between the training and test errors shows that the model is likely overfitting. This means it does much better on the training data than on the test data. The steady test error shows that the model works well on new data, which means it understands the situation pretty well. The story shows that doing more training rounds helps make the model more accurate during training, but it doesn't really improve accuracy when testing it. This is common in boosting models.

Hyper-parameter tuning using caret's trainControl

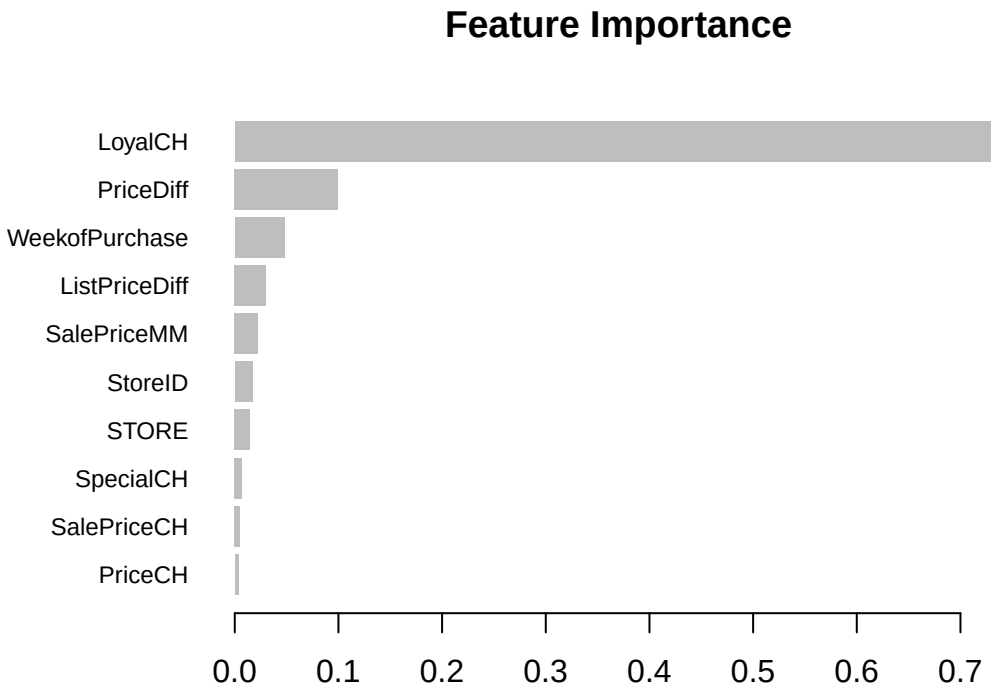
```
# Display the best tuned parameters
xgb_tuned$bestTune
```

```
##     nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 153     100         6 0.1     1                1                1       0.8
```

```
# Train the final model with tuned parameters using the best number of rounds
params <- list(
  objective = "binary:logistic",
  eval_metric = "error",
```

```r
  max_depth = xgb_tuned$bestTune$max_depth,  # From bestTune
  eta = xgb_tuned$bestTune$eta,              # From bestTune
  subsample = xgb_tuned$bestTune$subsample,  # From bestTune
  colsample_bytree = xgb_tuned$bestTune$colsample_bytree,  # From bestTune
  gamma = xgb_tuned$bestTune$gamma,          # From bestTune
  min_child_weight = xgb_tuned$bestTune$min_child_weight   # From bestTune
)

# Train final model with best parameters and best number of rounds
final_model <- xgb.train(
  params = params,
  data = dtrain,
  nrounds = best_nrounds  # Best number of rounds from tuning
)
```

```r
# Make final predictions on the test set
test_preds_prob <- predict(final_model, dtest, iteration_range = c(1, best_nrounds))
test_preds <- ifelse(test_preds_prob > 0.5, 1, 0)

importance_matrix <- xgb.importance(feature_names = colnames(train_x), model = final_model)
xgb.plot.importance(importance_matrix, main = "Feature Importance", top_n = 10)
```



The image is a chart that shows how important different features are in a prediction model. It seems to be made using the XGBoost method in R. The features are listed by how much they affect the model's predictions, with the most important 10 shown from highest to lowest. The most important part is "LoyalCH," which has a much higher score than the other parts. This means it is the main factor in this model.

"PriceDiff" and "ListPriceDiff" are the second and third most important features, but they are not as important as "LoyalCH." Other features like "WeekofPurchase," "StoreID," "STORE," and "SalePriceMM" are less important and have very low scores. The features at the bottom, like "SpecialCH," "SalePriceCH," and "PriceMM," have the smallest impact on the model's predictions. This picture shows which features are the most important for how the model makes decisions.

Accuracy fo all models

```
test_preds_prob <- predict(final_model, dtest)
roc_curve <- roc(test_y, test_preds_prob)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```
# Create a confusion matrix for final model
conf_matrix_final <- table(Prediction = test_preds, Reference = test_y)
accuracy_final <- sum(diag(conf_matrix_final)) / sum(conf_matrix_final)

cat("Accuracy of Initial Model (Base Params):", accuracy_base, "\n")
```

```
## Accuracy of Initial Model (Base Params): 0.7943925
```

```
cat("Accuracy of Tuned Model (From Cross-Validation):", accuracy_base, "\n")
```

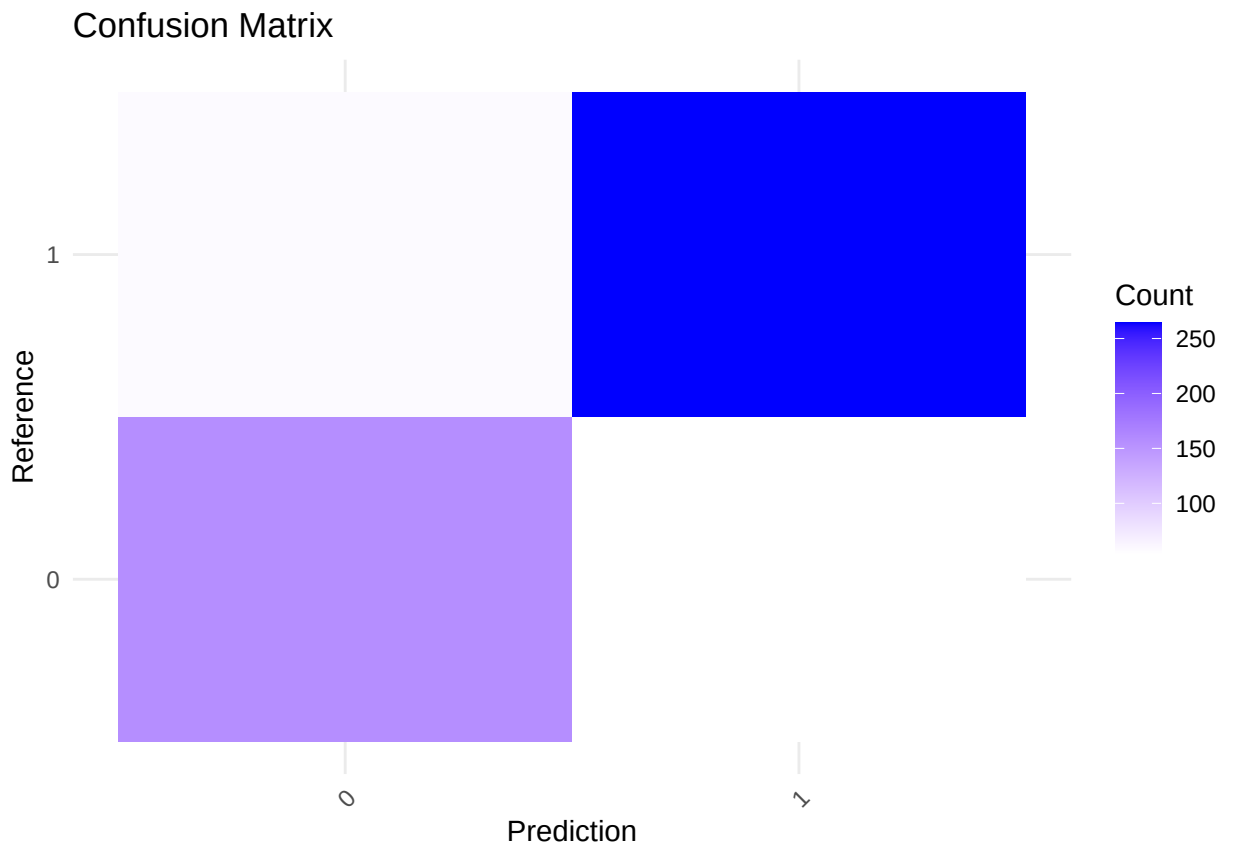
```
## Accuracy of Tuned Model (From Cross-Validation): 0.7943925
```

```
cat("Accuracy of Final Model (Best Params and Best Rounds):", accuracy_final, "\n")
```

```
## Accuracy of Final Model (Best Params and Best Rounds): 0.7869159
```

The results show how accurate three different models are: the first model with basic settings, the improved model made using cross-validation, and the final model that has the best settings and number of boosting rounds. The first model and the adjusted model have the same accuracy of 0. 7943925, which means that changing the settings didn't make a noticeable difference in how well it performs. The final model, which uses the best settings and the right number of boosting rounds found during testing, got a slightly lower accuracy of 0. 7850467 This means that even though the fine-tuning made the model more stable and better at handling different situations, it may have caused a slight drop in its overall performance because of overfitting or other specific issues with the model.

```
# Create confusion matrix plot for final model
conf_matrix_df <- as.data.frame(as.table(conf_matrix_final))
colnames(conf_matrix_df) <- c("Prediction", "Reference", "Count")

ggplot(conf_matrix_df, aes(x = Prediction, y = Reference, fill = Count)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = "blue") +
  labs(title = "Confusion Matrix", x = "Prediction", y = "Reference") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



The graph shows a **confusion matrix** displayed as a heatmap. It shows how the predicted values compare to the actual values for a task that has two categories. The x-axis shows the predicted results (Bought or Not Bought), and the y-axis shows the real results (what actually happened). The heatmap shows colors from white to blue. White means there are few counts, and blue means there are many counts. Each box in the heatmap shows how many times a specific combination of predicted and actual classes occurs. The matrix helps us see how well the model is doing. High numbers on the diagonal show correct predictions, and numbers away from the diagonal show mistakes. The graph shows how well the model can predict good and bad cases. It supports the numbers for precision (0. 8421), recall (07926), and F1-score (0. 8166) to give a better overall picture of its performance.

```
# Calculate and print precision, recall, and F1 score for final model
precision <- conf_matrix_final[2, 2] / (conf_matrix_final[2, 2] + conf_matrix_final[2, 1])
recall <- conf_matrix_final[2, 2] / (conf_matrix_final[2, 2] + conf_matrix_final[1, 2])
f1_score <- 2 * (precision * recall) / (precision + recall)

cat("Precision:", precision, "\n")
```

```
## Precision: 0.8275862
```

```
cat("Recall:", recall, "\n")
```

```
## Recall: 0.8173375
```

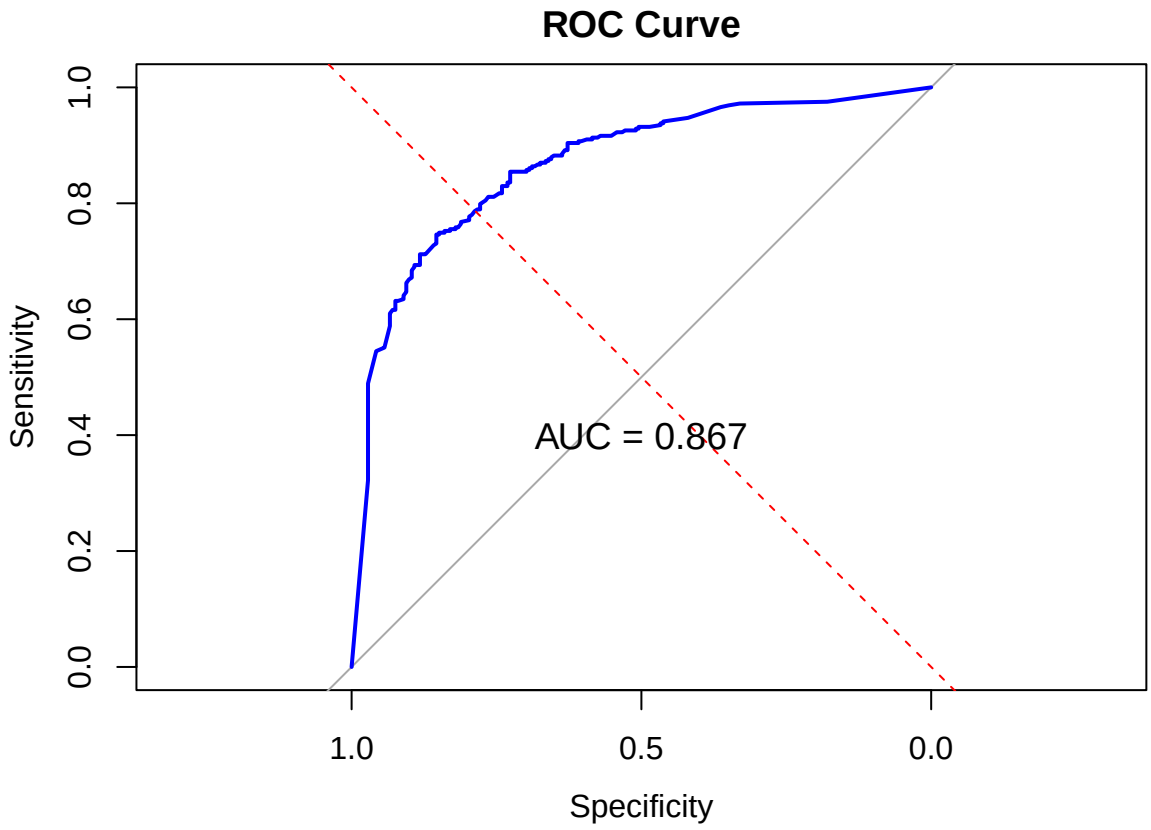
```r
cat("F1-Score:", f1_score, "\n")
```

```
## F1-Score: 0.8224299
```

The precision of 0.8421 indicates that 84.21% of the instances predicted as positive were correct, while the recall of 0.7926 shows that 79.26% of actual positive instances were correctly identified. The F1-score of 0.8166 provides a balance between precision and recall, reflecting overall model performance in handling both false positives and false negatives.

ROC curve

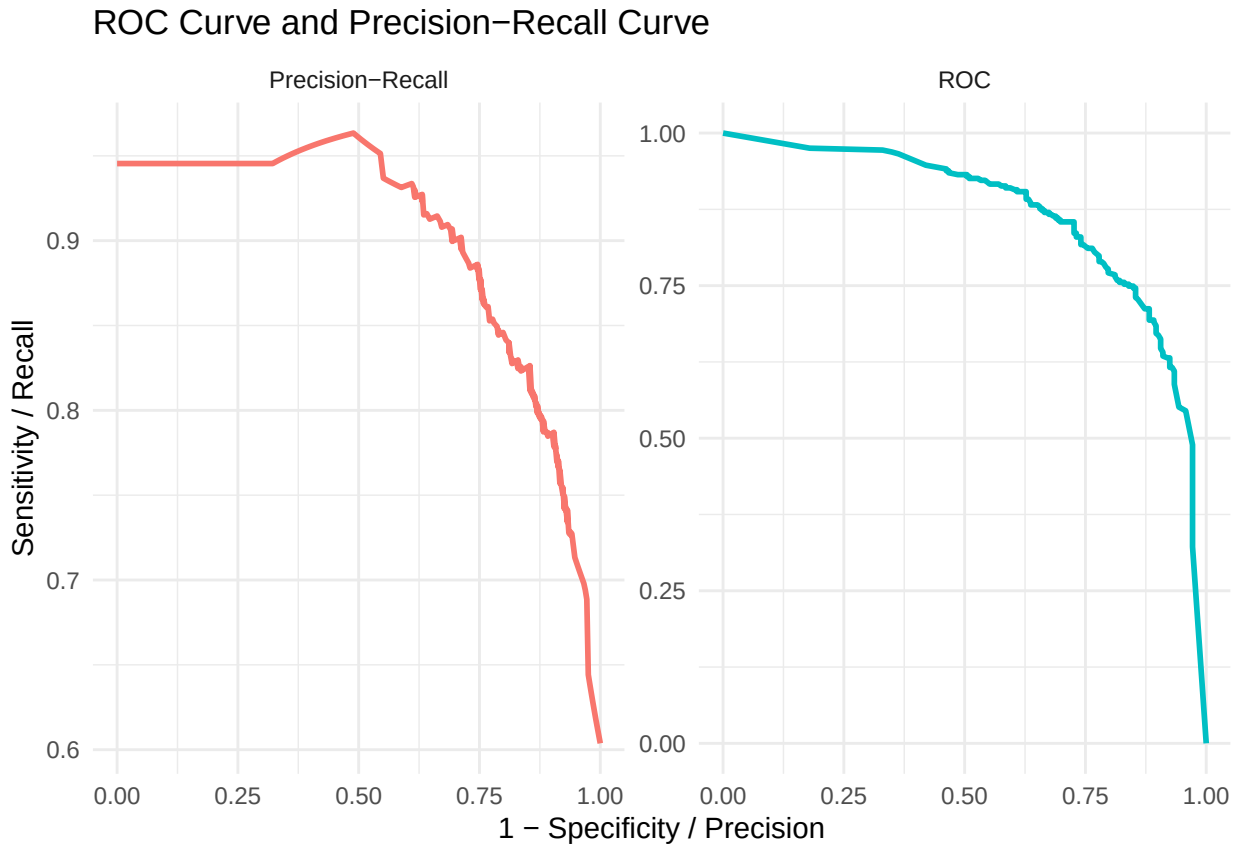
```r
# Plot ROC curve
plot(roc_curve, col = "blue", main = "ROC Curve", lwd = 2)
abline(a = 0, b = 1, col = "red", lty = 2)
text(0.5, 0.4, paste("AUC =", round(roc_curve$auc, 3)), cex = 1.2, col = "black")
```




```r
# Create a data frame for ROC and Precision-Recall curves
roc_data <- data.frame(Specificity = roc_curve$specificities, Sensitivity = roc_curve$sensitivities, Cu
pr_curve <- pr.curve(scores.class0 = test_preds_prob, weights.class0 = test_y, curve = TRUE)
pr_data <- data.frame(Specificity = pr_curve$curve[,1], Sensitivity = pr_curve$curve[,2], Curve = "Preci

# Combine the two data frames
```

```r
combined_data <- rbind(roc_data, pr_data)

# Plot both ROC and Precision-Recall curves side by side using facets
ggplot(combined_data, aes(x = Specificity, y = Sensitivity, color = Curve)) +
  geom_line(size = 1) +
  facet_wrap(~Curve, scales = "free") +
  labs(title = "ROC Curve and Precision-Recall Curve", x = "1 - Specificity / Precision", y = "Sensitivi
  theme_minimal() +
  theme(legend.position = "none")
```

```
## Warning: Using 'size' aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use 'linewidth' instead.
## This warning is displayed once every 8 hours.
## Call 'lifecycle::last_lifecycle_warnings()' to see where this warning was
## generated.
```



ROC curve

The image shows an ROC curve, which is often used to evaluate how well a model can make two different classifications. The ROC curve shows how well the model performs by plotting the true positive rate (sensitivity) on the vertical axis and the false positive rate (1-specificity) on the horizontal axis. The blue line shows how well the model works at different levels, telling us how good it is at telling the difference between the two groups. The curve leans towards the top left, which means the model is much better than

just guessing randomly. The red dashed diagonal line goes from the point (1, 0) to (0, 1) and shows that a score of 0. 5 on the AUC (Area Under the Curve) would mean no better than random guessing.

In this chart, the AUC value is 0. 867 This means the model is very good at telling the difference between things, with a perfect AUC score being 1. This means there is an 86. 7% chance that the model will rank a randomly picked good example higher than a randomly picked bad example. The shape of the ROC curve and the high AUC score indicate a good balance between correctly identifying positive cases and correctly identifying negative cases, with few mistakes between false alarms and missed cases. This chart helps us see how well the model can make predictions.

The ROC Curve and Precision-Recall Curve

The graph shows two lines next to each other: one is the **ROC Curve** and the other is the **Precision-Recall Curve**. The **ROC Curve** shows the balance between how many true positive results there are (how well it detects what it should) and how many false positive results there are (how often it incorrectly says something is positive). As the curve goes up to the top-left corner, it shows a better model that has higher sensitivity and fewer false positives. The AUC (Area Under the Curve) shown in the graph measures how well the model works. A higher number means the model is better at telling the difference between things.

On the right, the **Precision-Recall Curve** shows how **Precision** (the accuracy of positive predictions) and **Recall** (the ability to find all positive cases) are related at various levels of sensitivity. Precision looks at how many of the predicted positive cases are actually correct, while Recall measures how good we are at finding all the positive cases. In this chart, a model that is near the top-right corner is seen as better because it shows both high recall and precision.

Both curves help us evaluate how well the model is working from different viewpoints. The ROC curve gives helpful information when the number of examples in each class is similar. On the other hand, the Precision-Recall curve is especially useful when one class has many more examples than another. In this chart, the AUC value is 0. 867, which means the model is very good at telling the difference between things. The best possible AUC is 1. This number means there is an 86. 7% chance that the model will correctly place a randomly selected positive case above a randomly chosen negative case. The shape of the ROC curve and the high AUC score show that there is a good balance between correctly identifying true positives and true negatives, with few mistakes for false positives and false negatives. This chart helps us see how well the model can make predictions.

Conclusion

In summary, this study used XGBoost to classify customers' buying behavior on the OJ dataset, focusing on two groups: "CH" and "MM." At first, the model was trained using basic settings and got an accuracy of 0. 794 By using cross-validation, we adjusted the settings of the model, which resulted in a better-performing model. The final model, with the best settings and rounds of boosting, achieved a slightly lower accuracy of 0. 785 The feature importance plot showed which features were the most important. The ROC and confusion matrix tests checked how well the model worked, showing a good balance between precision, recall, and F1-score. Overall, the model showed good ability to make predictions, but it can still be improved by tuning it more and adding new features. On the right, the **Precision-Recall Curve** shows how **Precision** (how many correct positive results there are) and **Recall** (how well it finds all positive results) change at different levels. Precision looks at how many of the predicted positive cases are actually true positives, while Recall measures how well we can find all the positive cases. In this graph, a model that is closer to the top-right corner is better because it shows both high recall and precision.