Walmart Sales Forecasting

Minwoo Park
MIST 5635
University of Georgia
April 29, 2025

Table of Contents

1 Problem Formulation & Data Understanding

1.1 Context and Goal

- In the highly competitive retail industry, understanding customer purchasing behavior is crucial for sales as performance fluctuates due to various factors such as seasonal trends, customer demographics, and promotional events. To better understand purchasing patterns and enhance its decision-making, Walmart seeks to utilize e-commerce data to forecast individual purchase amounts based on customer and product information.

- The primary goal of this project is to **predict the purchase** amount that a customer is likely to spend, using demographic and product-related data. This is a **regression problem** where the target variable is the purchase amount in dollars.

- Machine learning techniques can help to predict Walmart's sales performance by learning patterns from historical customer demographics and product characteristics to predict future purchases. By training regression models on this data, it can anticipate spending behavior at the individual transaction level which leads to more informed business strategies.

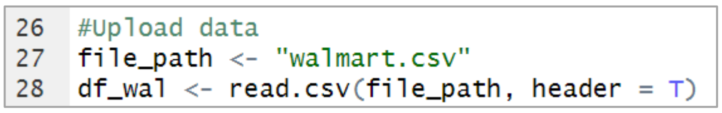
1.2 Data Description

- The dataset is sourced from Kaggle, titled as *"Walmart e-Commerce Sales Dataset"* (also known as the Black Friday Sales dataset). It provides rich insights into customer demographics, purchasing patterns, and product preference. It captures transaction-level purchase records from Walmart's online platform.

Variable	Description
User_ID	Unique identifier for each customer
Product_ID	Unique identifier for each product
Gender	Gender of the customer (M, F)
Age	Age range of the customer (e.g., 26-35, 36-45)
Occupation	Encoded occupation category
City_Category	Type of city the user resides in (A, B, or C)
Stay_In_Current_City_Years	Number of years customer has stayed in the city
Marital_Status	Marital status (0: single, 1: married)
Product_Category	Encoded product category
Purchase	Target variable: amount spent in the transaction (USD)

Table 1.1 Variables in Raw Data

For analysis, the dataset is initially uploaded and stored as *df_wal*.

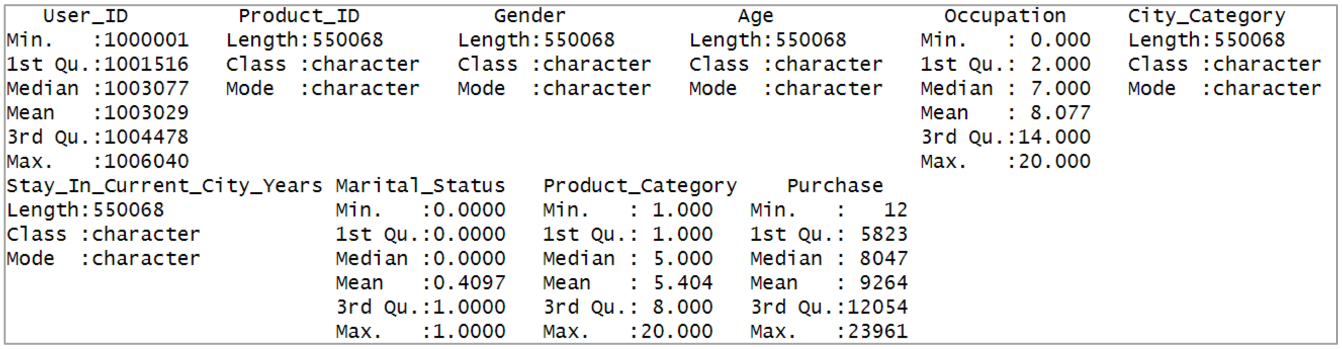


```r
#Upload data
file_path <- "walmart.csv"
df_wal <- read.csv(file_path, header = T)
```

Code Box 1.1: Data upload

1.3 Time Period / Context:

- The dataset does not include timestamps or exact dates. Code Box 1.2 shows a snapshot of Walmart's data summary, which provides a quick overview of the key variables and their respective statistics.

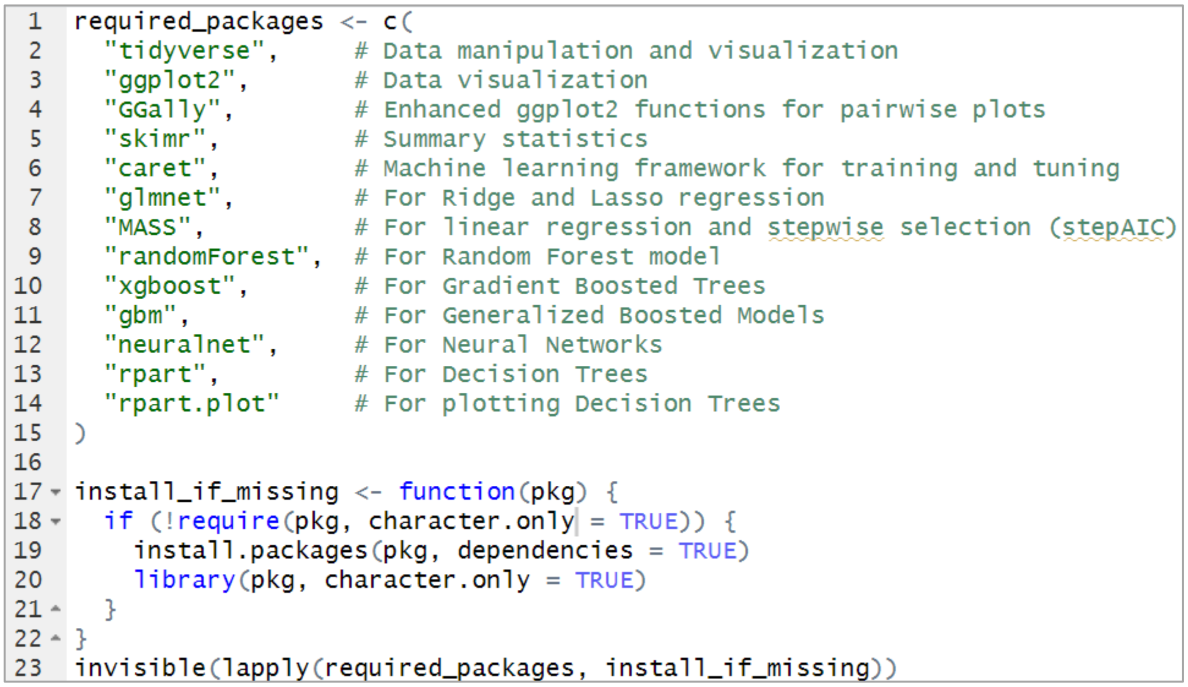


```
    User_ID          Product_ID           Gender              Age             Occupation      City_Category
Min.    :1000001   Length:550068      Length:550068      Length:550068      Min.   : 0.000   Length:550068
1st Qu.:1001516   Class :character   Class :character   Class :character   1st Qu.: 2.000   Class :character
Median :1003077   Mode  :character   Mode  :character   Mode  :character   Median : 7.000   Mode  :character
Mean   :1003029                                                             Mean   : 8.077
3rd Qu.:1004478                                                             3rd Qu.:14.000
Max.   :1006040                                                             Max.   :20.000
Stay_In_Current_City_Years Marital_Status   Product_Category    Purchase
Length:550068              Min.   :0.0000   Min.   : 1.000   Min.   :   12
Class :character           1st Qu.:0.0000   1st Qu.: 1.000   1st Qu.: 5823
Mode  :character           Median :0.0000   Median : 5.000   Median : 8047
                           Mean   :0.4097   Mean   : 5.404   Mean   : 9264
                           3rd Qu.:1.0000   3rd Qu.: 8.000   3rd Qu.:12054
                           Max.   :1.0000   Max.   :20.000   Max.   :23961
```

Code Box 1.2: summary(df_wal)

1.4 Libraries Used in the Project

- To conduct the analysis and build predictive models for Walmart's purchase prediction, I utilized several R libraries that provide efficient tools for data manipulation, modeling, and evaluation. The following libraries were used in the project:



```r
required_packages <- c(
  "tidyverse",      # Data manipulation and visualization
  "ggplot2",        # Data visualization
  "GGally",         # Enhanced ggplot2 functions for pairwise plots
  "skimr",          # Summary statistics
  "caret",          # Machine learning framework for training and tuning
  "glmnet",         # For Ridge and Lasso regression
  "MASS",           # For linear regression and stepwise selection (stepAIC)
  "randomForest",   # For Random Forest model
  "xgboost",        # For Gradient Boosted Trees
  "gbm",            # For Generalized Boosted Models
  "neuralnet",      # For Neural Networks
  "rpart",          # For Decision Trees
  "rpart.plot"      # For plotting Decision Trees
)

install_if_missing <- function(pkg) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg, dependencies = TRUE)
    library(pkg, character.only = TRUE)
  }
}
invisible(lapply(required_packages, install_if_missing))
```

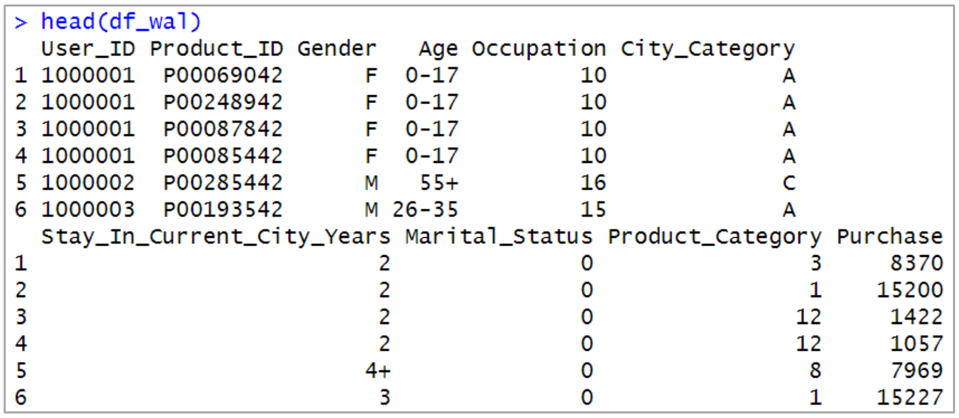
Code Box 1.3: Libraries for Project

- This list ensures that all necessary libraries are installed and loaded into the R environment before starting the analysis. It also helps anyone running the code to easily install the required dependencies if they are missing.

2
Data Cleaning & Exploration

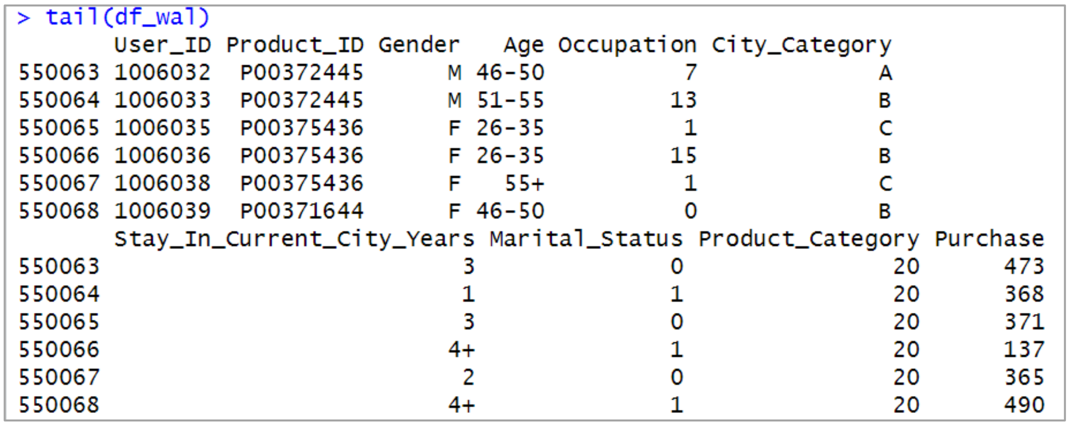
2.1 Viewing the Structure

- Checking head, and tail rows to understand the overall structure of the dataset.



```
> head(df_wal)
  User_ID Product_ID Gender   Age Occupation City_Category
1 1000001  P00069042      F  0-17         10             A
2 1000001  P00248942      F  0-17         10             A
3 1000001  P00087842      F  0-17         10             A
4 1000001  P00085442      F  0-17         10             A
5 1000002  P00285442      M   55+         16             C
6 1000003  P00193542      M 26-35         15             A
  Stay_In_Current_City_Years Marital_Status Product_Category Purchase
1                          2              0                3     8370
2                          2              0                1    15200
3                          2              0               12     1422
4                          2              0               12     1057
5                         4+              0                8     7969
6                          3              0                1    15227
```

Code Box 2.1: head(df_wal)

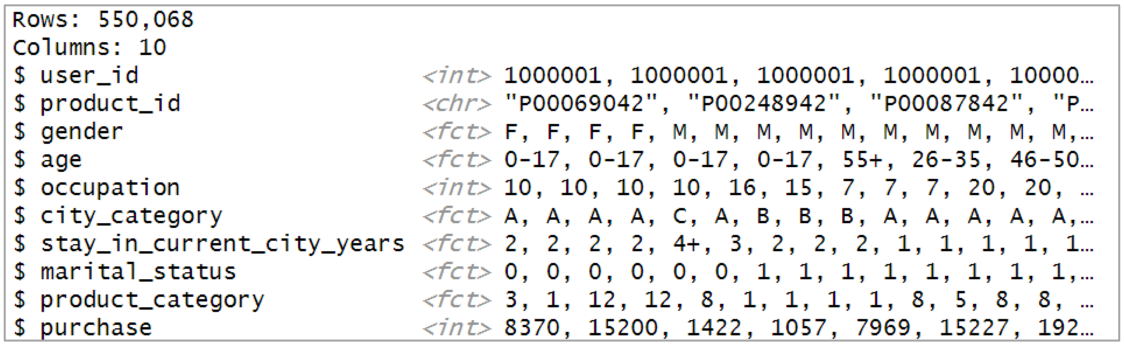


```
> tail(df_wal)
       User_ID Product_ID Gender   Age Occupation City_Category
550063 1006032  P00372445      M 46-50          7             A
550064 1006033  P00372445      M 51-55         13             B
550065 1006035  P00375436      F 26-35          1             C
550066 1006036  P00375436      F 26-35         15             B
550067 1006038  P00375436      F   55+          1             C
550068 1006039  P00371644      F 46-50          0             B
       Stay_In_Current_City_Years Marital_Status Product_Category Purchase
550063                          3              0               20      473
550064                          1              1               20      368
550065                          3              0               20      371
550066                         4+              1               20      137
550067                          2              0               20      365
550068                         4+              1               20      490
```

Code Box 2.2: tail(df_wal)

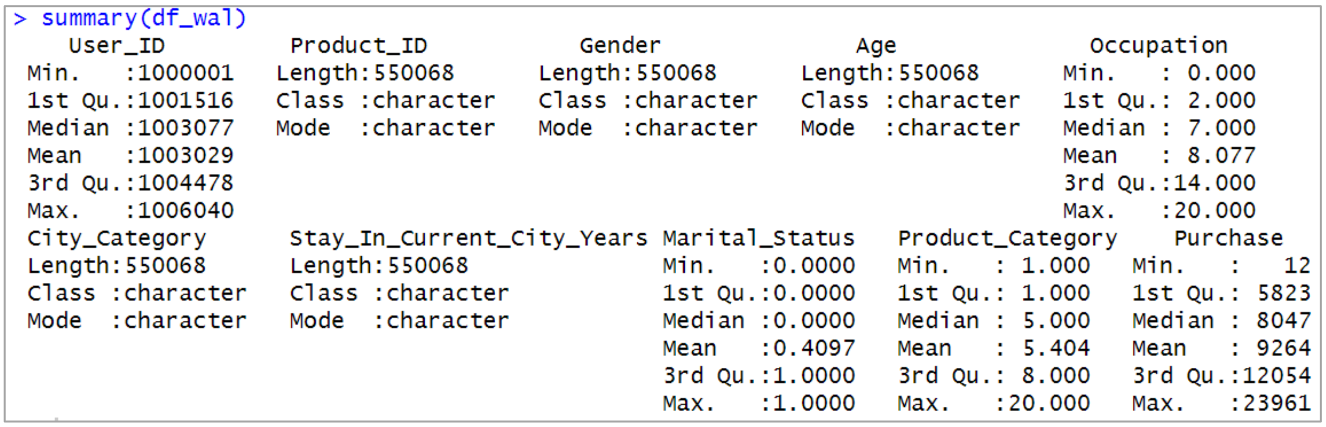
2.2 Data Types and Summary

- Used the following commands to examine the structure, types, and distributions of the variables:
  - *glimpse(df_wal)*: Provides a quick view of each column's type and sample values.
  - *skim(df_wal)*: Gives extended summary statistics (missing values, mean, etc.).
  - *summary(df_wal)*: Shows minimum, median, and maximum for numeric variables.



```
Rows: 550,068
Columns: 10
$ user_id                    <int> 1000001, 1000001, 1000001, 1000001, 10000…
$ product_id                 <chr> "P00069042", "P00248942", "P00087842", "P…
$ gender                     <fct> F, F, F, F, M, M, M, M, M, M, M, M, M, M,…
$ age                        <fct> 0-17, 0-17, 0-17, 0-17, 55+, 26-35, 46-50…
$ occupation                 <int> 10, 10, 10, 10, 16, 15, 7, 7, 7, 20, 20, …
$ city_category              <fct> A, A, A, A, C, A, B, B, B, A, A, A, A, A,…
$ stay_in_current_city_years <fct> 2, 2, 2, 2, 4+, 3, 2, 2, 2, 1, 1, 1, 1, 1…
$ marital_status             <fct> 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,…
$ product_category           <fct> 3, 1, 12, 12, 8, 1, 1, 1, 1, 8, 5, 8, 8, …
$ purchase                   <int> 8370, 15200, 1422, 1057, 7969, 15227, 192…
```

Code Box 2.3: glimpse(df_wal)



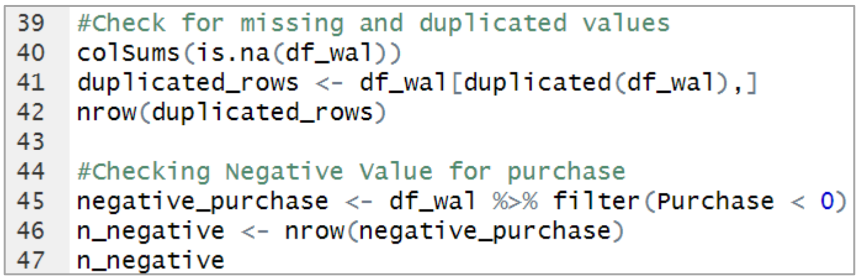
```
> summary(df_wal)
    User_ID          Product_ID           Gender              Age              Occupation    
 Min.   :1000001   Length:550068      Length:550068      Length:550068      Min.   : 0.000  
 1st Qu.:1001516   Class :character   Class :character   Class :character   1st Qu.: 2.000  
 Median :1003077   Mode  :character   Mode  :character   Mode  :character   Median : 7.000  
 Mean   :1003029                                                            Mean   : 8.077  
 3rd Qu.:1004478                                                            3rd Qu.:14.000  
 Max.   :1006040                                                            Max.   :20.000  
 City_Category      Stay_In_Current_City_Years Marital_Status   Product_Category    Purchase    
 Length:550068      Length:550068              Min.   :0.0000   Min.   : 1.000   Min.   :   12  
 Class :character   Class :character           1st Qu.:0.0000   1st Qu.: 1.000   1st Qu.: 5823  
 Mode  :character   Mode  :character           Median :0.0000   Median : 5.000   Median : 8047  
                                               Mean   :0.4097   Mean   : 5.404   Mean   : 9264  
                                               3rd Qu.:1.0000   3rd Qu.: 8.000   3rd Qu.:12054  
                                               Max.   :1.0000   Max.   :20.000   Max.   :23961  
```

Code Box 2.4: summary(df_wal)

2.3 Data Quality Check

- Performed a data quality check on the Walmart dataset:
    - **Missing Values**: No missing values were detected across any columns.
    - **Duplicated Rows**: No duplicated rows were found in the dataset.
    - **Negative Purchase Values**: No negative purchase amounts were identified.

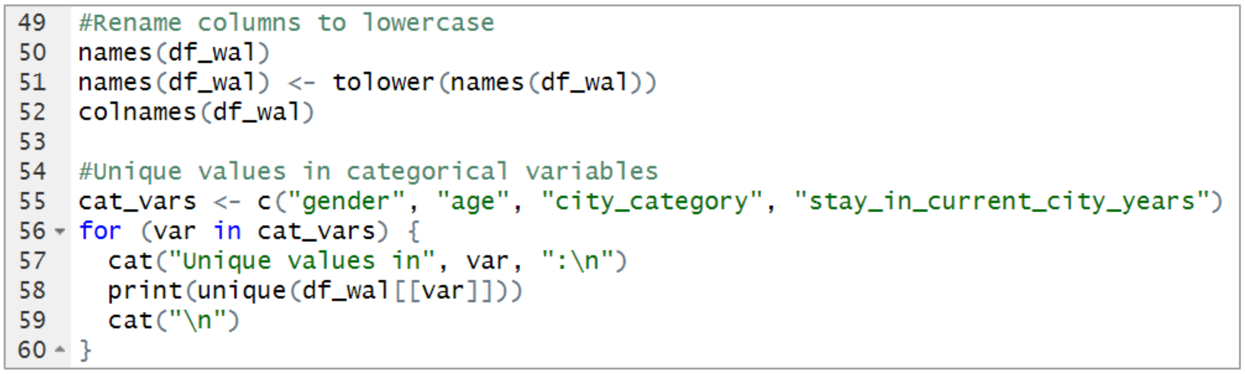
- The dataset is clean and free from any missing, duplicated, or invalid purchase amounts, which means it is ready for analysis.



```r
#Check for missing and duplicated values
colSums(is.na(df_wal))
duplicated_rows <- df_wal[duplicated(df_wal),]
nrow(duplicated_rows)

#Checking Negative Value for purchase
negative_purchase <- df_wal %>% filter(Purchase < 0)
n_negative <- nrow(negative_purchase)
n_negative
```

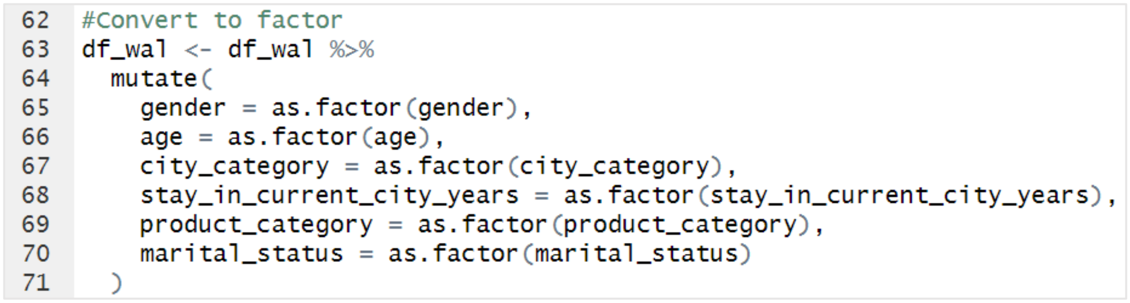
Code Box 2.5: Quality Check

- Additionally, to ensure consistency:
    - All column names were converted to lowercase.
    - Checked unique values in categorical variables
    - Categorical variables (gender, age, city_category, stay_in_current_city_years, product_category, and marital_status) were properly converted into factor types for analysis.



```r
#Rename columns to lowercase
names(df_wal)
names(df_wal) <- tolower(names(df_wal))
colnames(df_wal)

#Unique values in categorical variables
cat_vars <- c("gender", "age", "city_category", "stay_in_current_city_years")
for (var in cat_vars) {
  cat("Unique values in", var, ":\n")
  print(unique(df_wal[[var]]))
  cat("\n")
}
```

Code Box 2.6: Lower Casing and Checking Unique Values



```
#Convert to factor
df_wal <- df_wal %>%
  mutate(
    gender = as.factor(gender),
    age = as.factor(age),
    city_category = as.factor(city_category),
    stay_in_current_city_years = as.factor(stay_in_current_city_years),
    product_category = as.factor(product_category),
    marital_status = as.factor(marital_status)
  )
```

Code Box 2.7: Categorical Variables to factors

2.4 Exploratory Data Analysis (EDA)

- To better understand the relationships between features and the target variable (Purchase), I performed exploratory data analysis using various visualizations (*Figure 2.1 - 2.4*):
  - **Age vs Purchase**: The average purchase amount by different age groups, segmented by gender.
  - **Occupation vs Purchase**: The average purchase amount across various occupations, segmented by gender.
  - **Marital Status vs Purchase**: Distribution of purchase amounts for different marital statuses.
  - **Product Category vs Purchase (by Age)**: The average purchase amount across product categories, segmented by age groups.

- From these visualizations, it was observed that:
  - Each predictor variable shows an indirect relationship with the target variable.
  - Different demographic and product-related factors appear to influence the average purchase amount.

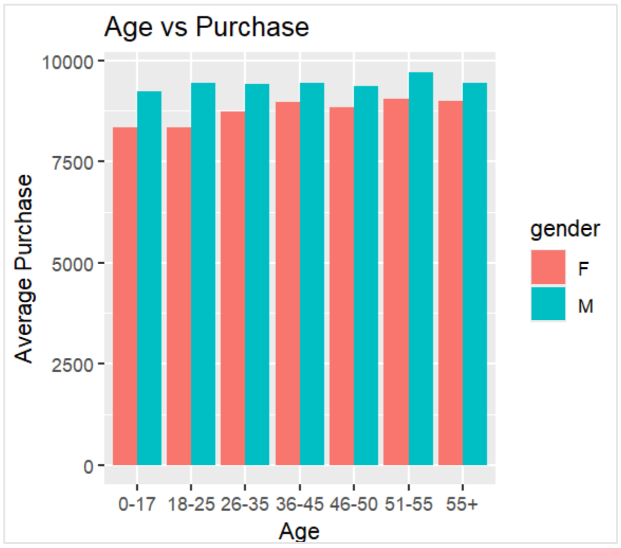


Figure 2.1: Age vs Purchase

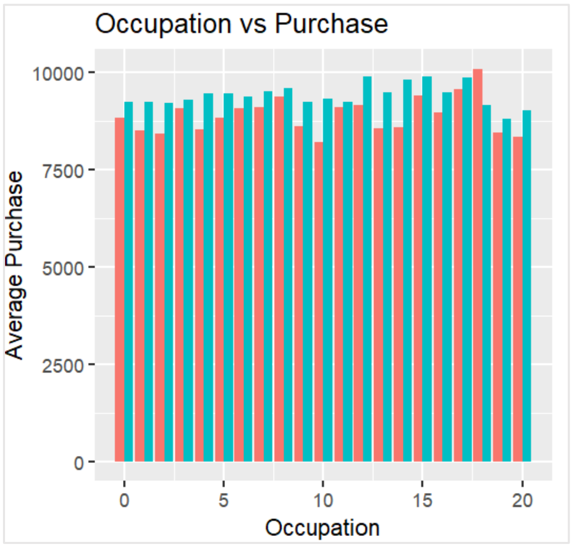
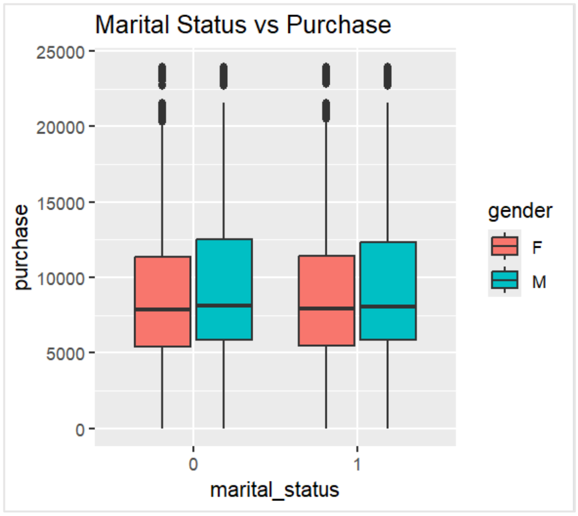


Figure 2.2 Occupation vs Purchase



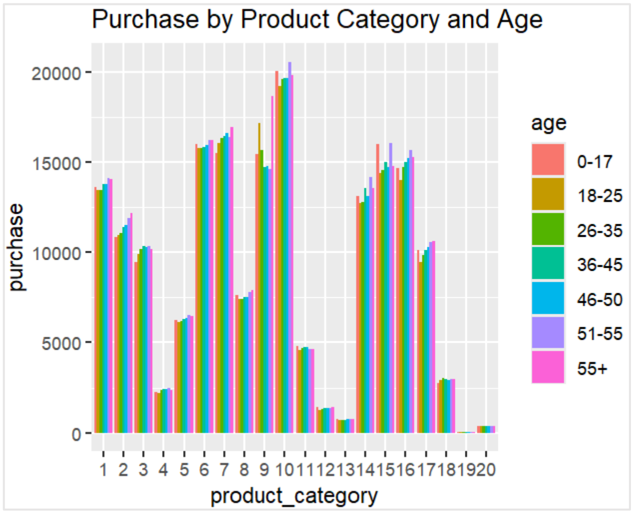


Figure 2.3: Marital Status vs Purchase

Figure 2.4 Product Category vs Purchase (Age)

2.5 Outlier Detection

- To better understand the distribution of the target variable (Purchase), I plotted a histogram (*Figure 2.5*):

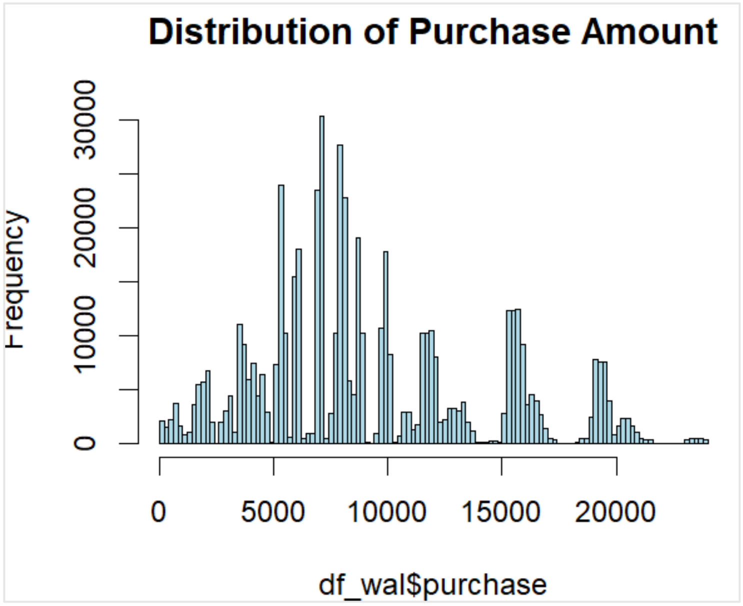


Figure 2.5: Distribution of Purchase Amount

- A few additional insights can be drawn from the graph:
  - **Concentration of Distribution**: Most transactions are concentrated around certain mid-range values (e.g., between $5,000 and $10,000).
  - **Unusual Distribution Shape**: The histogram shows several distinct peaks, indicating varying customer purchasing patterns.
  - **Right-skewed Distribution**: The histogram appears to be right-skewed. It means that most purchases are relatively low, but there are some transactions with high amounts.

- Given this, the histogram shows an asymmetric distribution with multiple peaks, indicating concentrated purchasing patterns across various customer segments. This also points to potential outliers, as the high-value purchases could be considered extreme or outlying transactions.

- As a following step, I created a **boxplot** for the purchase amounts (*Figure 2.6*):

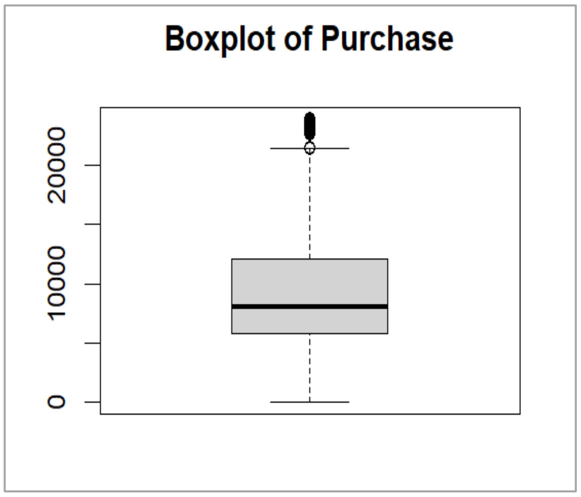


Figure 2.6: Box plot of Purchase

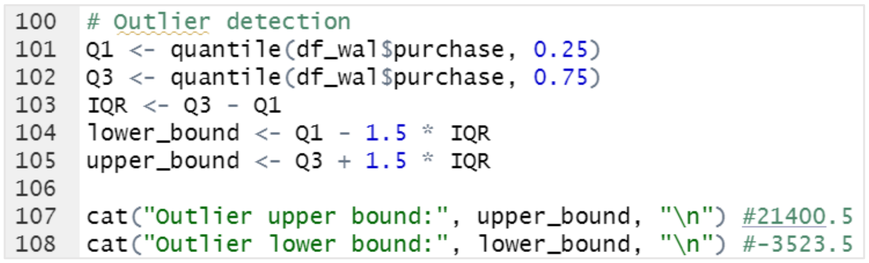
- The boxplot reveals that while the majority of purchases are within a typical range, there are several points above the upper whisker.

- Based on this observation, I proceeded with formal **outlier detection** using the Interquartile Range (IQR) method:
    - Outliers were defined as values outside the range: [Q1−1.5×IQR, Q3+1.5×IQR], where IQR = Q3 - Q1

- Based on the IQR method:
    - **Q1 (25%)**: $5823
    - **Q3 (75%)**: $12054
    - **IQR**: 6231
    - **Upper Bound**: 21400.5
    - **Lower Bound**: -3523.5

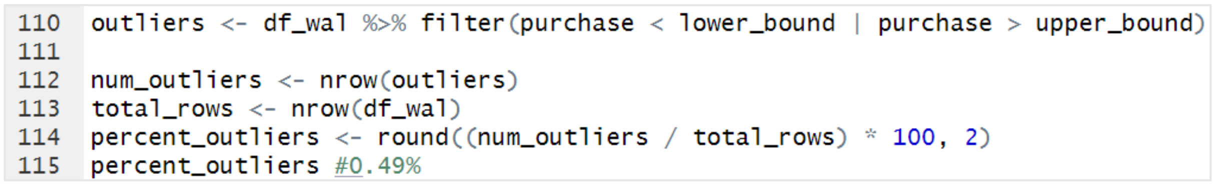
- Insight from the dataset:
- **Total Rows**: 550,068
- **Number of Outliers**: 2,677
- **Unique Users**: 5891
- **Unique Products**: 3631

- By overseeing Code Box 2.8-10 and Table 2, I conducted IQR (Interquartile Range) analysis to identify outliers. I calculated the percentage of outliers in the dataset, where the outliers amounted to 2,677 rows out of 550,068, which is 0.49%. The outlier purchases were made by 25.2% of all users, and the outlier products accounted for 1.3% of all products.



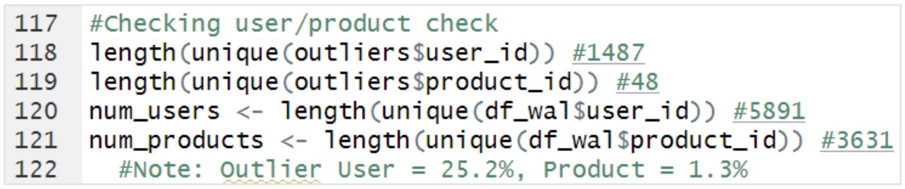
```
# Outlier detection
Q1 <- quantile(df_wal$purchase, 0.25)
Q3 <- quantile(df_wal$purchase, 0.75)
IQR <- Q3 - Q1
lower_bound <- Q1 - 1.5 * IQR
upper_bound <- Q3 + 1.5 * IQR

cat("Outlier upper bound:", upper_bound, "\n") #21400.5
cat("Outlier lower bound:", lower_bound, "\n") #-3523.5
```

Code Box 2.8: Outlier Detection Using IQR



```
outliers <- df_wal %>% filter(purchase < lower_bound | purchase > upper_bound)

num_outliers <- nrow(outliers)
total_rows <- nrow(df_wal)
percent_outliers <- round((num_outliers / total_rows) * 100, 2)
percent_outliers #0.49%
```

Code Box 2.9: Outlier Detection Calculation



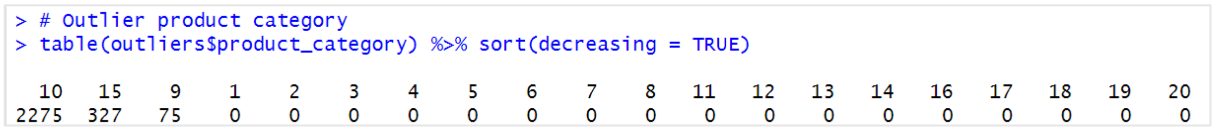
```
#Checking user/product check
length(unique(outliers$user_id)) #1487
length(unique(outliers$product_id)) #48
num_users <- length(unique(df_wal$user_id)) #5891
num_products <- length(unique(df_wal$product_id)) #3631
  #Note: Outlier User = 25.2%, Product = 1.3%
```

Code Box 2.10: Checking Outliers in User and Product

Metric	**Value**	**Interpretation**
Outliers among all transactions	**0.49%** (2,677 rows)	High-value purchases occurred rarely on a per-transaction basis
Outlier Users among all Users	**25.2%** (1,487 users)	A large portion of users made high-value purchases at least once
Outlier Products among all Products	**1.3%** (48 products)	High-value purchases occurred only in a small number of premium products

Table 2.1: Outlier Analysis Summary

- As seen in Code Box 2.11, it was found that the outliers were limited to product categories 10, 15, and 9.



```
> # Outlier product category
> table(outliers$product_category) %>% sort(decreasing = TRUE)

  10   15    9    1    2    3    4    5    6    7    8   11   12   13   14   16   17   18   19   20 
2275  327   75    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 
```

Code Box 2.11: Outliers by Product Category

- By synthesizing the data, it appears that the outliers are generally associated with a limited set of high-priced items. Although outliers account for only 0.49% of the total transactions, they involve 25.2% of all users and are concentrated within just 1.3% of all products. Furthermore, the outlier transactions are primarily concentrated in three product categories, suggesting that these represent typical premium products rather than random anomalies.

- However, since linear models such as Linear Regression are sensitive to outliers, I plan to apply a logarithmic transformation to the Purchase variable to stabilize variance for these models. For other models, I will proceed using the raw Purchase values, but in some cases, I will also experiment with log-transformed values to compare outcomes and assess whether the transformation improves model robustness.

3
Feature Engineering

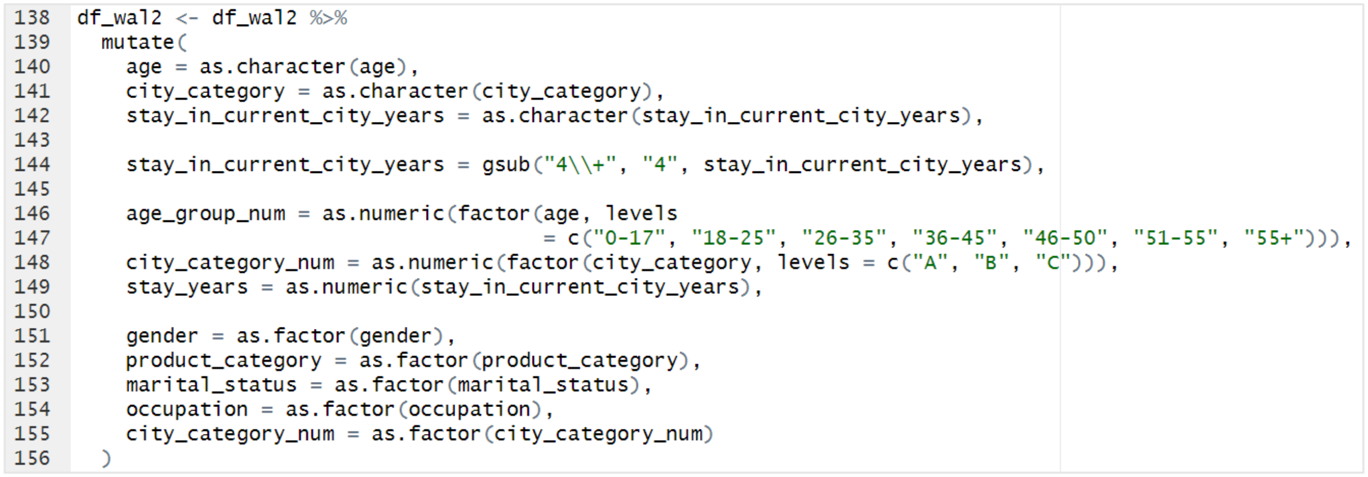
Overview

- Before conducting feature engineering, the original dataset *df_wal* was copied to a new dataset *df_wal2* to preserve the original data integrity.
- The feature engineering process included two major steps: (1) transformation and removal of unnecessary columns, and (2) feature encoding to prepare categorical variables for modeling.

3.1 Feature Creation and Transformation

- To enhance model performance and prepare the dataset for training, I conducted feature creation and transformation steps, focusing on reformatting categorical variables and removing unnecessary identifiers.

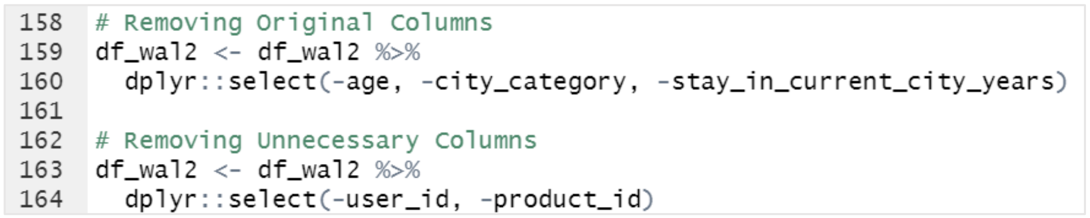
- **Transformation of Categorical Variables**:
    - The age, city_category, and stay_in_current_city_years variables were initially of type character and were converted to appropriate numeric forms.
    - For stay_in_current_city_years, the "4+" category was recoded as "4" to enable numeric conversion, resulting in a new variable stay_years.
    - A new variable age_group_num was created by encoding age groups in an ordinal manner (e.g., "0-17" = 1, "18-25" = 2, etc.).
    - Similarly, a new variable city_category_num was created by encoding city categories (A = 1, B = 2, C = 3).



```r
df_wal2 <- df_wal2 %>%
  mutate(
    age = as.character(age),
    city_category = as.character(city_category),
    stay_in_current_city_years = as.character(stay_in_current_city_years),

    stay_in_current_city_years = gsub("4\\+", "4", stay_in_current_city_years),

    age_group_num = as.numeric(factor(age, levels
                                = c("0-17", "18-25", "26-35", "36-45", "46-50", "51-55", "55+"))),
    city_category_num = as.numeric(factor(city_category, levels = c("A", "B", "C"))),
    stay_years = as.numeric(stay_in_current_city_years),

    gender = as.factor(gender),
    product_category = as.factor(product_category),
    marital_status = as.factor(marital_status),
    occupation = as.factor(occupation),
    city_category_num = as.factor(city_category_num)
  )
```

Code Box 3.1: Transformation of Categorical Variables

- **Removal of Unnecessary Columns**:
    - The user_id and product_id columns were removed from the dataset as they serve only as identifiers and do not contribute meaningful information for prediction.



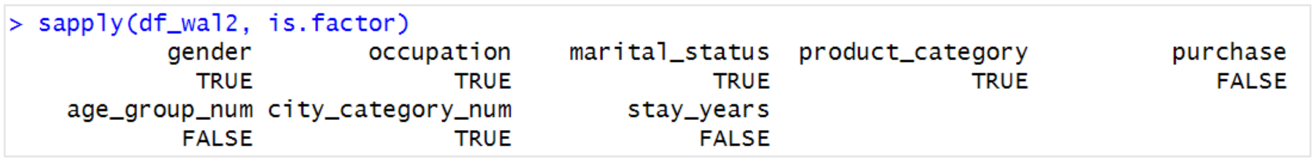
```r
# Removing Original Columns
df_wal2 <- df_wal2 %>%
  dplyr::select(-age, -city_category, -stay_in_current_city_years)

# Removing Unnecessary Columns
df_wal2 <- df_wal2 %>%
  dplyr::select(-user_id, -product_id)
```

Code Box 3.2: Removing Unnecessary Columns

3.2 Feature Encoding & Checking

- After transformation, I encoded categorical variables to ensure proper treatment during modeling.

- **Categorical Encoding**:
  - Variables such as gender, product_category, marital_status, occupation, and city_category_num were converted into factor types to ensure that modeling algorithms treat them as categorical variables.
  - After encoding, I additionally verified that each variable was correctly converted to the factor type, ensuring the dataset was properly prepared for modeling. The results of this verification can be found in Code Box 3.3.



```
> sapply(df_wal2, is.factor)
          gender        occupation    marital_status  product_category          purchase
            TRUE              TRUE              TRUE              TRUE             FALSE
   age_group_num city_category_num        stay_years
           FALSE              TRUE             FALSE
```

Code Box 3.3: Factor Type Verification after Feature Engineering

- These steps ensured that both numeric and categorical variables were correctly formatted, maintaining data integrity while preparing the dataset for training machine learning models.

4
Model Selection & Training

4.1 Overview

- To build a robust predictive model for purchase amounts, I explored multiple modeling approaches, applying both the original purchase values and a log-transformed version (log_purchase) to mitigate the impact of outliers and stabilize variance.

4.2 Models Considered (7 models):

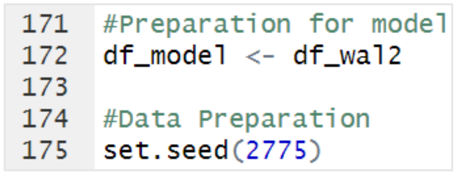
1. **Linear Regression**: A simple, interpretable baseline model for understanding linear relationships between predictors and the target.
2. **Ridge Regression**: Applied to prevent overfitting by adding a penalty for large coefficients in the presence of multicollinearity.
3. **Lasso Regression**: Used to enforce sparsity and select significant features by shrinking some coefficients to zero.
4. **Random Forest**: An ensemble learning model for capturing non-linear relationships, robust to outliers, and highly effective without requiring feature scaling.
5. **Gradient Boosting Machine (GBM)**: A powerful boosting technique that builds trees sequentially, correcting errors made by previous trees.
6. **XGBoost**: An optimized version of GBM, highly efficient and effective for large datasets with added regularization.
7. **Neural Network:** A flexible model capable of capturing complex non-linear patterns by learning hierarchical feature representations, suitable for high-dimensional data when properly tuned.

- Given the nature of Walmart's purchase data, which may exhibit both linear and non-linear characteristics, a diverse set of models was considered to ensure robust performance across different relationship patterns. *Table 4.1* summarizes the rationale for selecting each model, along with their key strengths and weaknesses.

- Each model considered in this project has specific hyperparameters that can be tuned to optimize performance. *Table 4.2* outlines the key hyperparameters for each model, along with the tuning approach and evaluation method used during training.

- To begin the modeling process, the cleaned dataset df_wal2 was assigned to a new object *df_model*.

- Additionally, to ensure the reproducibility of model training, data splitting, and cross-validation procedures, a random seed value of 2775 was set throughout the analysis (*Code Box 4.1*).



```
#Preparation for model
df_model <- df_wal2

#Data Preparation
set.seed(2775)
```

Code Box 4.1: Data Preparation and Reproducibility Setup

Model	Reason for Choosing	Strengths	Weaknesses
Linear Regression	Simple and interpretable baseline for linear relationships	Easy to interpret; good for linear trends	Sensitive to outliers and heteroscedasticity
Ridge Regression	Prevents overfitting and handles multicollinearity	Controls large coefficients; stable	No feature selection
Lasso Regression	Automatic feature selection and regularization	Feature selection; reduces complexity	May drop correlated features; needs scaling
Random Forest	Captures non-linear relationships; robust to outliers	Handles non-linearity, outliers; no scaling needed	Black-box; computationally heavy
GBM	Sequential error correction and flexible modeling	High predictive power; captures complex patterns	Sensitive to hyperparameters; prone to overfitting
XGBoost	Optimized GBM; regularization for better generalization	Efficient; handles large data; regularization included	Complex to tune
Neural Network	Captures highly complex non-linear relationships	Learns complex feature interactions	Needs tuning; prone to overfitting; less interpretable

Table 4.1: Summary of Model Selection, Strengths, and Weaknesses

Model	Key Hyperparameters Tuned	Tuning Method	Evaluation Method
Linear Regression	– (no hyperparameters)	–	5-fold cross-validation
Ridge Regression	lambda (L2 regularization strength)	Grid search 10^3 to 10^{-3}	5-fold cross-validation
Lasso Regression	lambda (L1 regularization strength)	Grid search 10^3 to 10^{-3}	5-fold cross-validation
Random Forest	mtry (number of variables tried at each split), ntree (number of trees)	Grid search; manual adjustment for ntree	5-fold CV → train/test split (for speed)
GBM	n.trees, interaction.depth, shrinkage, n.minobsinnode	Grid search	train/test split (70/30)
XGBoost	eta (learning rate), max_depth, nrounds, subsample, colsample_bytree	Grid search	train/test split (70/30)
Neural Network	hidden layer size, learning rate, number of epochs	Manual tuning (or grid search if applied)	train/test split (70/30)

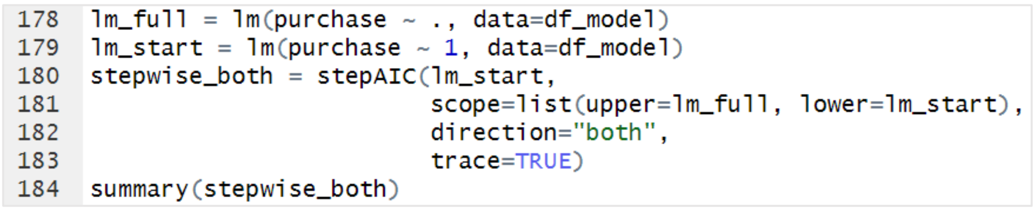
Table 4.2: Hyperparameter Tuning and Evaluation Summary

- As shown in Table 4.2, 5-fold cross-validation was initially applied to models such as Linear, Ridge, and Lasso regression to ensure stable and accurate performance estimation.
- However, for models like Random Forest and beyond, each training iteration exceeded one hour due to the dataset's large size (over 500,000 records) and hardware limitations.
- Therefore, to ensure practical feasibility, a 70/30 train-test split using createDataPartition() was used instead of cross-validation for those models.

4.3 Stepwise Feature Selection

- Before applying various modeling techniques, a stepwise selection approach was employed to identify an optimal subset of variables to include in the models, starting with a linear regression framework. Specifically, a stepwise selection based on the Akaike Information Criterion (AIC) was performed using both forward and backward directions, beginning from a null model (purchase ~ 1) and considering the full model (purchase ~ .).

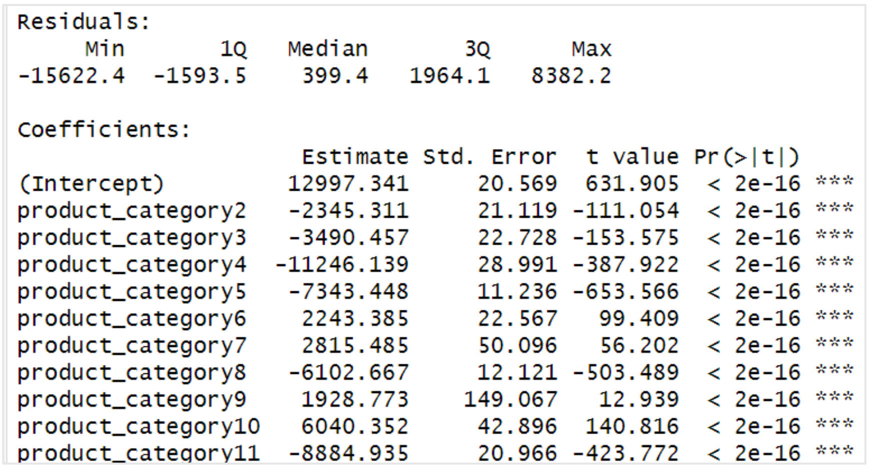
- This process aimed to refine the feature set and enhance model generalization by selecting variables most strongly associated with the target variable, purchase amount. (*Code Box 4.2*)



```
lm_full = lm(purchase ~ ., data=df_model)
lm_start = lm(purchase ~ 1, data=df_model)
stepwise_both = stepAIC(lm_start,
                        scope=list(upper=lm_full, lower=lm_start),
                        direction="both",
                        trace=TRUE)
summary(stepwise_both)
```

Code Box 4.2: Stepwise Selection

- Based on the summary of the stepwise selection results (*Code Box 4.3*), it was observed that almost all features demonstrated low p-values, indicating strong statistical significance in relation to the purchase amount.
- Therefore, for the subsequent model training and evaluation, the full feature set will be used rather than a reduced subset, ensuring that all available information is leveraged across different modeling approaches.

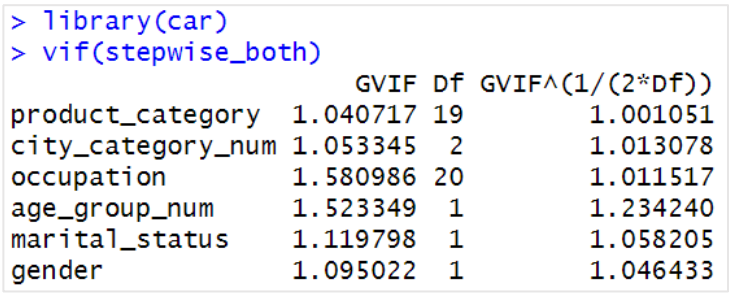


```
Residuals:
     Min       1Q   Median       3Q      Max 
-15622.4  -1593.5    399.4   1964.1   8382.2 

Coefficients:
                     Estimate Std. Error  t value Pr(>|t|)    
(Intercept)         12997.341     20.569  631.905  < 2e-16 ***
product_category2   -2345.311     21.119 -111.054  < 2e-16 ***
product_category3   -3490.457     22.728 -153.575  < 2e-16 ***
product_category4  -11246.139     28.991 -387.922  < 2e-16 ***
product_category5   -7343.448     11.236 -653.566  < 2e-16 ***
product_category6    2243.385     22.567   99.409  < 2e-16 ***
product_category7    2815.485     50.096   56.202  < 2e-16 ***
product_category8   -6102.667     12.121 -503.489  < 2e-16 ***
product_category9    1928.773    149.067   12.939  < 2e-16 ***
product_category10   6040.352     42.896  140.816  < 2e-16 ***
product_category11  -8884.935     20.966 -423.772  < 2e-16 ***
```

Code Box 4.3: Sample of Stepwise Result

4.4 Variance Inflation Factor (VIF)

- Additionally, to ensure that multicollinearity would not adversely affect model stability, a Variance Inflation Factor (VIF) analysis was conducted on the selected features.
- As shown in *Code Box 4.4*, all features exhibited VIF values well below 5, indicating a low risk of multicollinearity.
- Thus, it was confirmed that the selected features were appropriate for modeling without concerns about redundancy or instability.



```
> library(car)
> vif(stepwise_both)
                     GVIF Df GVIF^(1/(2*Df))
product_category 1.040717 19        1.001051
city_category_num 1.053345  2       1.013078
occupation       1.580986 20        1.011517
age_group_num    1.523349  1        1.234240
marital_status   1.119798  1        1.058205
gender           1.095022  1        1.046433
```

Code Box 4.4: Multicollinearity Check using VIF

- Now we are ready for train & evaluation models.
- Since model training and evaluation were conducted concurrently for each algorithm, detailed performance metrics and analysis are presented together in Section 5 (Model Evaluation & Comparison).

5

Model Evaluation & Comparison

5.1 Evaluation Metrics Overview

- Before comparing the performance of different models, it is important to define the evaluation metrics used for this regression task.

- Since the objective is to predict Walmart customers' purchase amounts, three widely accepted error metrics were applied: RMSE (Root Mean Squared Error), MAE (Mean Absolute Error), and MAPE (Mean Absolute Percentage Error).

- These metrics allow for comprehensive assessment of model accuracy, robustness to outliers, and interpretability in business contexts. A brief summary of each metric is shown in *Table 5.1*.

Metric	Full Name	Interpretation	Strengths / Usage
RMSE	Root Mean Squared Error	Measures average magnitude of squared errors	Penalizes large errors heavily; emphasizes accuracy
MAE	Mean Absolute Error	Measures average absolute deviation from actuals	Easy to interpret; robust to extreme outliers
MAPE	Mean Absolute Percentage Error	Measures average percentage deviation from actuals	Useful for business decision-making; scale-independent

Table 5.1: Evaluation Metrics Used for Model Comparison

5.2 Model Training and Evaluation Results

5.2.1 Linear Regression

- Linear regression was selected as a baseline model because Walmart customers' purchase behavior could potentially exhibit linear patterns with respect to the available features.

5.2.1.1 Residual Diagnostics

- Before proceeding with full model training and evaluation for linear regression, I conducted a residual analysis to assess the variance structure and check the appropriateness of applying a standard linear model to the purchase data.
- Specifically, the residuals from the initial full linear regression model were plotted against the fitted values to visually inspect patterns such as heteroscedasticity (non-constant variance).

- **Residual Plot for Raw Purchase Values**
    - As shown in *Figure 5.1*, when modeling the original purchase values without any transformation, the residuals exhibited a clear pattern of heteroscedasticity, where the spread of residuals increased with the fitted values.
    - This suggests that the model's variance is not constant across the range of predictions, likely due to the influence of outliers and the skewed distribution of purchase amounts.

- **Residual Plot for Log-Transformed Purchase Values**
    - As shown in *Figure 5.1*, after applying a log transformation to the purchase values, the residuals exhibited a much more stable variance across the range of fitted values.
    - The heteroscedasticity issue observed in the raw model was largely mitigated, supporting the appropriateness of using the log-transformed target for subsequent modeling.

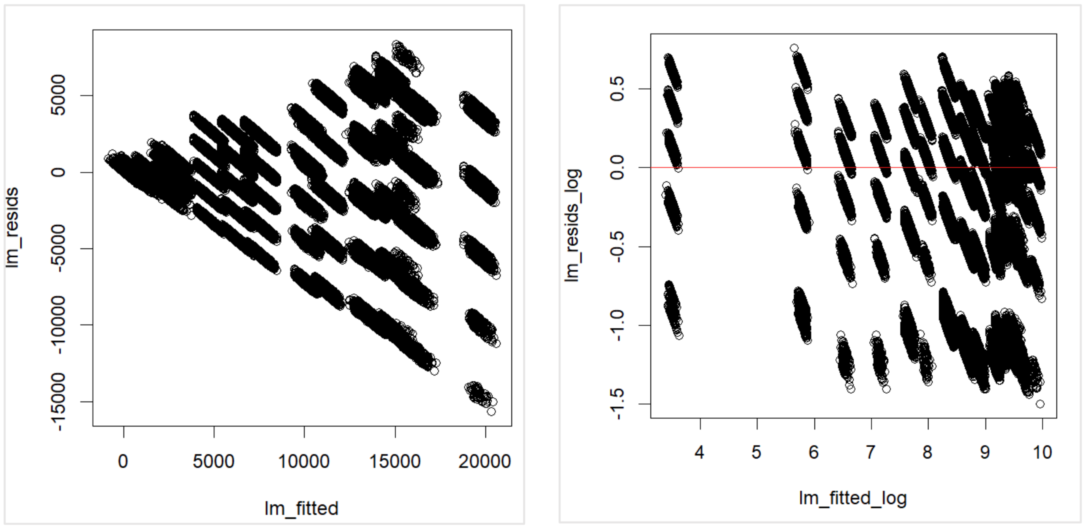


Figure 5.1: Residual Plots Before and After Log Transformation of Purchase Values

- The Q-Q plot after applying the log transformation (Figure 5.2) shows that the residuals align reasonably well with the theoretical normal line in the middle range.

- Although the raw purchase residuals appear visually aligned with the normal line in the Q-Q plot, the <u>extremely large residual scale</u> indicates instability and high sensitivity to outliers.

- After log transformation, while minor deviations persist at the tails, the overall variance was significantly stabilized, and residuals in the central range aligned more closely with the theoretical normal distribution, leading to a more reliable modeling foundation.

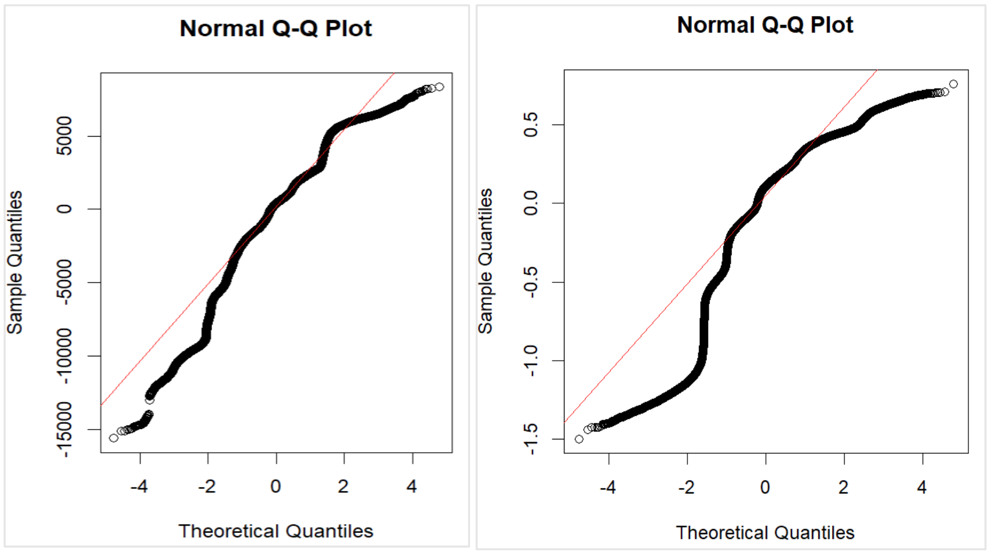


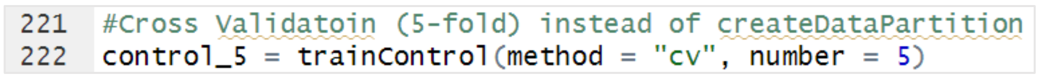
Figure 5.2: Q-Q Plot of Residuals Before and After Log Transformation

- This improvement justified the use of log-transformed targets for more stable training and better generalization in subsequent models.

5.2.1.2 Cross-Validation Strategy for Linear Regression

- To ensure robust performance evaluation and reduce overfitting, 5-fold cross-validation was used during the training of the linear regression model. (*Code Box 5.1*)

- This technique splits the dataset into five equal parts, training the model on four parts and validating it on the remaining one, iterating the process five times.



```
#Cross Validatoin (5-fold) instead of createDataPartition
control_5 = trainControl(method = "cv", number = 5)
```

Code Box 5.1: Cross-Validation with 5-fold

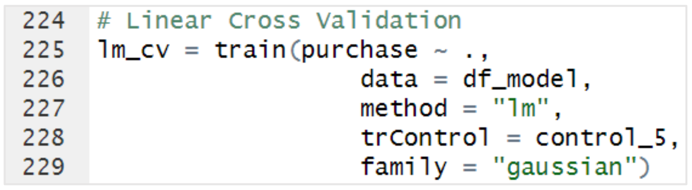
5.2.1.3 Linear Regression Model for Raw Purchase Values

- Using the caret package, the linear regression model was trained with 5-fold cross-validation as defined in *Code Box 5.2*.

- The resulting performance summary (*Code Box 5.3*) showed:
    - RMSE: 3014.089
    - R-squared: 0.639946
    - MAE: 2282.591

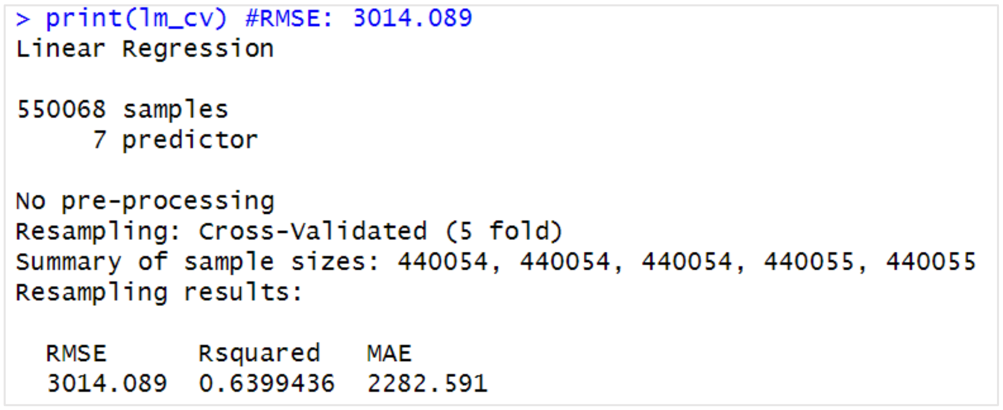
- This relatively high RMSE value suggests that the model's performance is negatively affected by the presence of outliers in the raw purchase amounts.

- The influence of extreme purchase values likely caused greater residual errors, reducing the overall model stability and accuracy.



```
# Linear Cross Validation
lm_cv = train(purchase ~ .,
                   data = df_model,
                   method = "lm",
                   trControl = control_5,
                   family = "gaussian")
```

Code Box 5.2: Linear Regression Training Setup for Raw Purchase Values



```
> print(lm_cv) #RMSE: 3014.089
Linear Regression

550068 samples
     7 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 440054, 440054, 440054, 440055, 440055
Resampling results:

  RMSE      Rsquared   MAE
  3014.089  0.6399436  2282.591
```

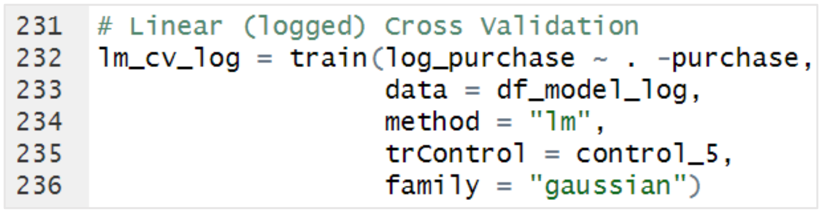
Code Box 5.3: Linear Regression Cross-Validation Results for Raw Purchase Values

5.2.1.4 Linear Regression Model After Log Transformation of Purchase Values

- Using the caret package, the linear regression model was trained with 5-fold cross-validation as defined in *Code Box 5.4*.

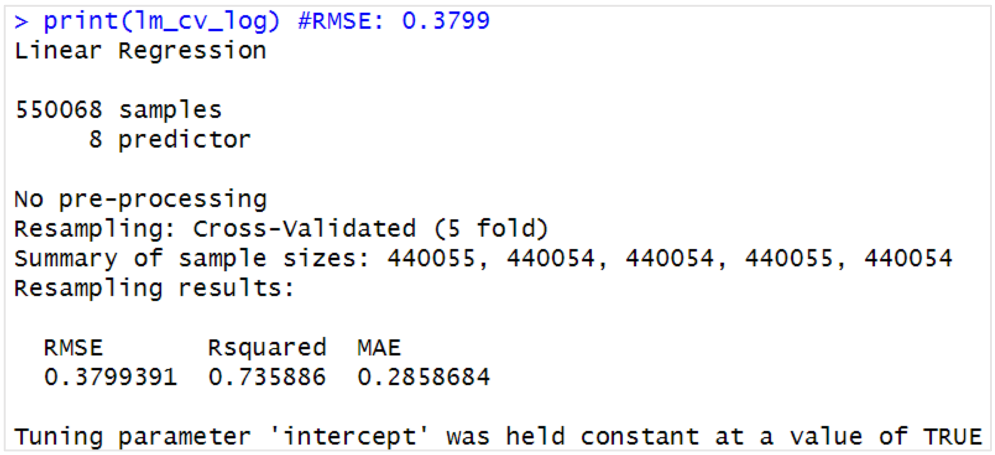
- The cross-validation results are summarized in *Code Box 5.5*, where the RMSE was significantly reduced to 0.3799 after applying the log transformation.

- This improvement indicates that the log transformation effectively stabilized variance and mitigated the influence of outliers, leading to enhanced model stability and predictive performance.



```
# Linear (logged) Cross Validation
lm_cv_log = train(log_purchase ~ . -purchase,
                  data = df_model_log,
                  method = "lm",
                  trControl = control_5,
                  family = "gaussian")
```

Code Box 5.4: Linear Regression Training Setup for Log-Transformed Purchase Values



```
> print(lm_cv_log) #RMSE: 0.3799
Linear Regression

550068 samples
     8 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 440055, 440054, 440054, 440055, 440054
Resampling results:

  RMSE       Rsquared   MAE
  0.3799391  0.735886   0.2858684

Tuning parameter 'intercept' was held constant at a value of TRUE
```

Code Box 5.5: Linear Regression Cross-Validation Results for Log-Transformed Purchase Values

5.2.1.5 Current Findings with Linear Regression and Linear Regression (Log-Transformed):

- As summarized in *Table 5.2*, the Log-Transformed Linear Regression model demonstrated the best overall performance among the models trained so far.

- The Log-Transformed Linear Regression model achieved the lowest RMSE (0.3799), showing significantly better stability and predictive performance compared to the raw model (RMSE 3014.089).

- The R-squared value of 0.7359 indicates that the model explains approximately 73.6% of the variance in the purchase amounts.

- The MAE (Mean Absolute Error) is 0.2859, meaning that on average, the model's predictions deviate from the actual purchase amounts by approximately 0.29.

- Furthermore, the strong performance of a linear model, especially after log transformation, suggests that there is an underlying <u>linear relationship</u> between the predictors and the log-transformed purchase amounts.

Model	RMSE	R-squared	MAE
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	-	-	-
Ridge Regression (Log)	-	-	-
Lasso Regression (Raw)	-	-	-
Lasso Regression (Log)	-	-	-
Random Forest (Raw)	-	-	-
Random Forest (Log)	-	-	-
GBM (Raw)	-	-	-
GBM (Log-Transformed)	-	-	-
XGBoost (Raw)	-	-	-
XGBoost (Log-Transformed)	-	-	-
Neural Network (Log)	-	-	-

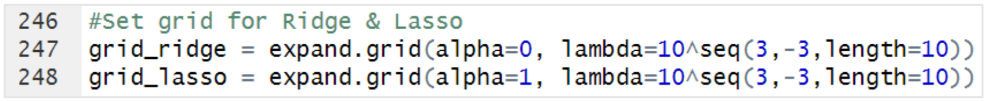
Table 5.2: Current Model Performance Summary (Linear Regression Models)

5.2.2 Ridge Regression

- Ridge regression was selected to improve prediction accuracy while controlling for potential multicollinearity among predictors, which is common in customer and product datasets like Walmart's.
- By applying an L2 regularization penalty, Ridge regression helps prevent overfitting and ensures that the model generalizes better to unseen data.

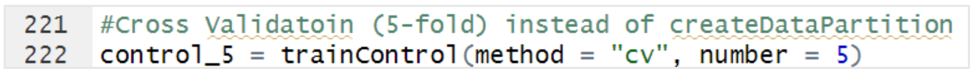
5.2.2.1 Hyperparameter Utilization for Ridge

- To optimize the Ridge regression model, I applied hyperparameter tuning using grid search and 5-fold cross-validation.
    - As shown in *Code Box 5.6*, a parameter grid was created for the Ridge model by fixing alpha = 0 (which indicates Ridge regression) and tuning the regularization parameter lambda over a wide range from 10^3 to 10^-3
    - In addition, *Code Box 5.7* illustrates the use of 5-fold cross-validation via trainControl(method = "cv", number = 5) to robustly evaluate model performance across different lambda values.
- These techniques ensured that the model was both well-regularized and appropriately validated, minimizing the risk of overfitting while maintaining generalization capability.



```
#Set grid for Ridge & Lasso
grid_ridge = expand.grid(alpha=0, lambda=10^seq(3,-3,length=10))
grid_lasso = expand.grid(alpha=1, lambda=10^seq(3,-3,length=10))
```

Code Box 5.6: Grid Search for Ridge and Lasso

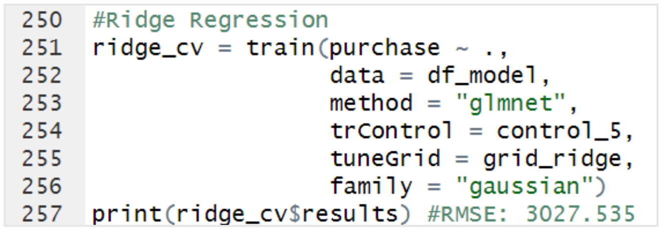


```
#Cross Validatoin (5-fold) instead of createDataPartition
control_5 = trainControl(method = "cv", number = 5)
```

Code Box 5.7: 5-Fold Cross-Validation Setup

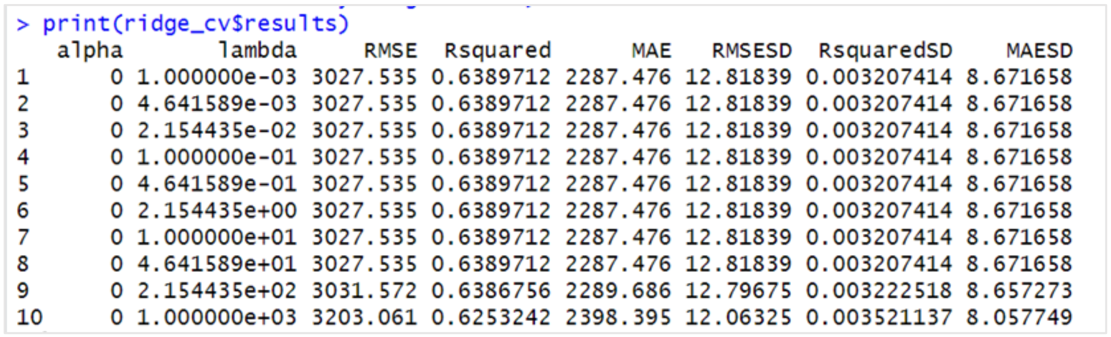
5.2.2.2 Ridge Regression Model for Raw Purchase Values

- Using the 5-fold cross-validation setup and the grid defined in Code Boxes 5.6 and 5.7, I trained a Ridge regression model on purchase values. The model performance was evaluated across different λ values, and the results including RMSE, R-squared, and MAE are shown in Code Box 5.9. The best RMSE achieved was approximately 3027.535.



```
#Ridge Regression
ridge_cv = train(purchase ~ .,
                 data = df_model,
                 method = "glmnet",
                 trControl = control_5,
                 tuneGrid = grid_ridge,
                 family = "gaussian")
print(ridge_cv$results) #RMSE: 3027.535
```

Code Box 5.8: Training of Ridge Regression with Raw Purchase

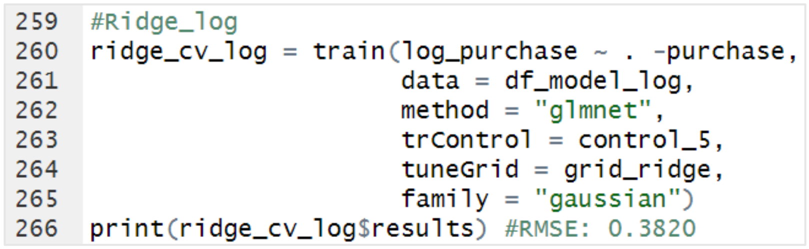


```
> print(ridge_cv$results)
   alpha       lambda     RMSE  Rsquared      MAE   RMSESD  RsquaredSD    MAESD
1      0 1.000000e-03 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
2      0 4.641589e-03 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
3      0 2.154435e-02 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
4      0 1.000000e-01 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
5      0 4.641589e-01 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
6      0 2.154435e+00 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
7      0 1.000000e+01 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
8      0 4.641589e+01 3027.535 0.6389712 2287.476 12.81839 0.003207414 8.671658
9      0 2.154435e+02 3031.572 0.6386756 2289.686 12.79675 0.003222518 8.657273
10     0 1.000000e+03 3203.061 0.6253242 2398.395 12.06325 0.003521137 8.057749
```

Code Box 5.9: Result of Ridge Regression with Raw Purchase

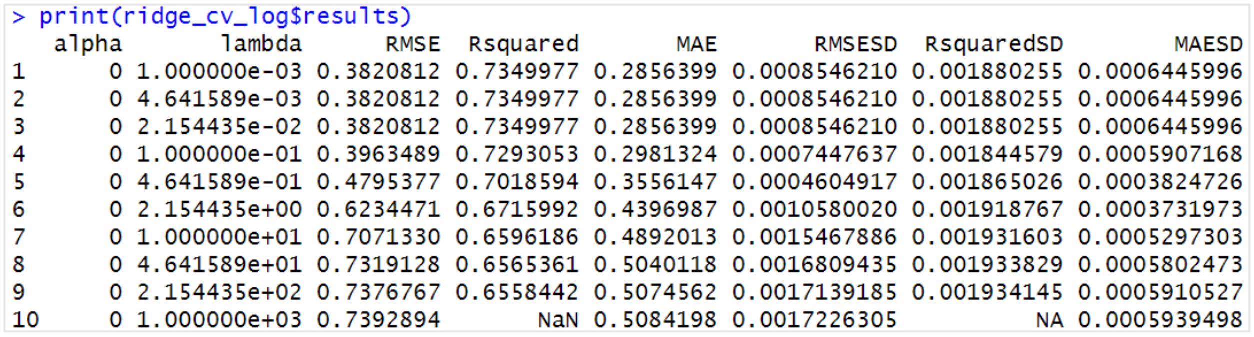
5.2.2.3 Ridge Regression Model After Log Transformation of Purchase Values

- Using the 5-fold cross-validation setup and the grid defined in Code Boxes 5.6 and 5.7, I trained a Ridge regression model on the log-transformed purchase values. The model performance was evaluated across different λ values, and the results including RMSE, R-squared, and MAE are shown in Code Box 5.11. The best RMSE achieved was approximately 0.3820.



```
#Ridge_log
ridge_cv_log = train(log_purchase ~ . -purchase,
                     data = df_model_log,
                     method = "glmnet",
                     trControl = control_5,
                     tuneGrid = grid_ridge,
                     family = "gaussian")
print(ridge_cv_log$results) #RMSE: 0.3820
```

Code Box 5.10: Training Ridge Regression with Logged Purchase



```
> print(ridge_cv_log$results)
   alpha       lambda      RMSE  Rsquared       MAE       RMSESD  RsquaredSD        MAESD
1      0 1.000000e-03 0.3820812 0.7349977 0.2856399 0.0008546210 0.001880255 0.0006445996
2      0 4.641589e-03 0.3820812 0.7349977 0.2856399 0.0008546210 0.001880255 0.0006445996
3      0 2.154435e-02 0.3820812 0.7349977 0.2856399 0.0008546210 0.001880255 0.0006445996
4      0 1.000000e-01 0.3963489 0.7293053 0.2981324 0.0007447637 0.001844579 0.0005907168
5      0 4.641589e-01 0.4795377 0.7018594 0.3556147 0.0004604917 0.001865026 0.0003824726
6      0 2.154435e+00 0.6234471 0.6715992 0.4396987 0.0010580020 0.001918767 0.0003731973
7      0 1.000000e+01 0.7071330 0.6596186 0.4892013 0.0015467886 0.001931603 0.0005297303
8      0 4.641589e+01 0.7319128 0.6565361 0.5040118 0.0016809435 0.001933829 0.0005802473
9      0 2.154435e+02 0.7376767 0.6558442 0.5074562 0.0017139185 0.001934145 0.0005910527
10     0 1.000000e+03 0.7392894       NaN 0.5084198 0.0017226305          NA 0.0005939498
```

Code Box 5.11: Result of Ridge Regression with Logged Purchase

5.2.1.4 Current Findings with Ridge/Logged Ridge Regression

- In Ridge Regression using raw purchase data, the model's regularization effect was limited due to the large scale of values and the strong influence of outliers. As the λ value increased, performance tended to degrade because of over-penalization.

- In contrast, applying Ridge Regression to log-transformed purchase values helped mitigate the impact of outliers and normalize the distribution, allowing the regularization to function more effectively.

- However, even in the log-transformed model, excessively large λ values led to over-regularization, resulting in a loss of predictive power.

Model	**RMSE**	**R-squared**	**MAE**
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	3027.535	0.638971	2287.476
Ridge Regression (Log)	0.3820	0.734997	0.285639
Lasso Regression (Raw)	-	-	-
Lasso Regression (Log)	-	-	-
Random Forest (Raw)	-	-	-
Random Forest (Log)	-	-	-
GBM (Raw)	-	-	-
GBM (Log-Transformed)	-	-	-
XGBoost (Raw)	-	-	-
XGBoost (Log-Transformed)	-	-	-
Neural Network (Log)	-	-	-

Table 5.3: Current Model Performance Summary (Ridge Regression Models)

5.2.3 Lasso Regression

- Lasso regression applies an L1 regularization penalty, which enables automatic variable selection by shrinking the coefficients of less important features to zero. This results in a simplified model and helps identify the most relevant predictors. Given Walmart’s diverse customer and product features, Lasso was selected as well-suited for isolating key factors driving purchase behavior.

5.2.3.1 Hyperparameter Utilization for Lasso

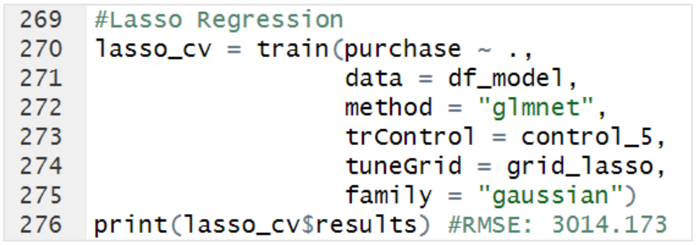
- To optimize the Lasso regression model, I applied hyperparameter tuning using grid search and 5-fold cross-validation.

    - As shown in *Code Box 5.6*, a parameter grid was created for the Ridge model by fixing alpha = 0 (which indicates Ridge regression) and tuning the regularization parameter lambda over a wide range from 10^3 to 10^-3
    - In addition, *Code Box 5.7* illustrates the use of 5-fold cross-validation via trainControl(method = "cv", number = 5) to robustly evaluate model performance across different lambda values.

- These techniques ensured that the model was both well-regularized and appropriately validated, minimizing the risk of overfitting while maintaining generalization capability.

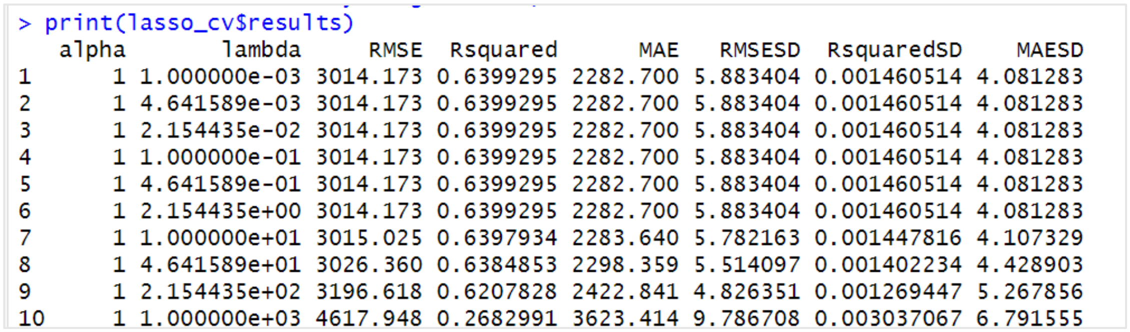
5.2.3.2 Lasoo Regression Model for Raw Purchase Values

- Using the 5-fold cross-validation setup and the grid defined in Code Boxes 5.6 and 5.7, I trained a Lasso regression model on the log-transformed purchase values. The model performance was evaluated across different λ values, and the results including RMSE, R-squared, and MAE are shown in Code Box 5.13. The best RMSE achieved was approximately 3014.173



```r
#Lasso Regression
lasso_cv = train(purchase ~ .,
                 data = df_model,
                 method = "glmnet",
                 trControl = control_5,
                 tuneGrid = grid_lasso,
                 family = "gaussian")
print(lasso_cv$results) #RMSE: 3014.173
```

Code Box 5.12: Training Lasso Regression with Raw Purchase

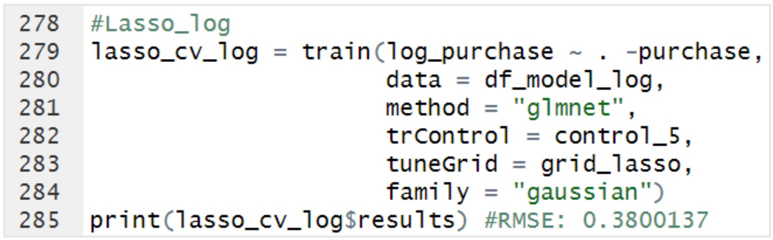


```
> print(lasso_cv$results)
   alpha       lambda     RMSE  Rsquared      MAE   RMSESD  RsquaredSD    MAESD
1      1 1.000000e-03 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
2      1 4.641589e-03 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
3      1 2.154435e-02 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
4      1 1.000000e-01 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
5      1 4.641589e-01 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
6      1 2.154435e+00 3014.173 0.6399295 2282.700 5.883404 0.001460514 4.081283
7      1 1.000000e+01 3015.025 0.6397934 2283.640 5.782163 0.001447816 4.107329
8      1 4.641589e+01 3026.360 0.6384853 2298.359 5.514097 0.001402234 4.428903
9      1 2.154435e+02 3196.618 0.6207828 2422.841 4.826351 0.001269447 5.267856
10     1 1.000000e+03 4617.948 0.2682991 3623.414 9.786708 0.003037067 6.791555
```

Code Box 5.13: Result of Lasso Regression with Logged Purchase

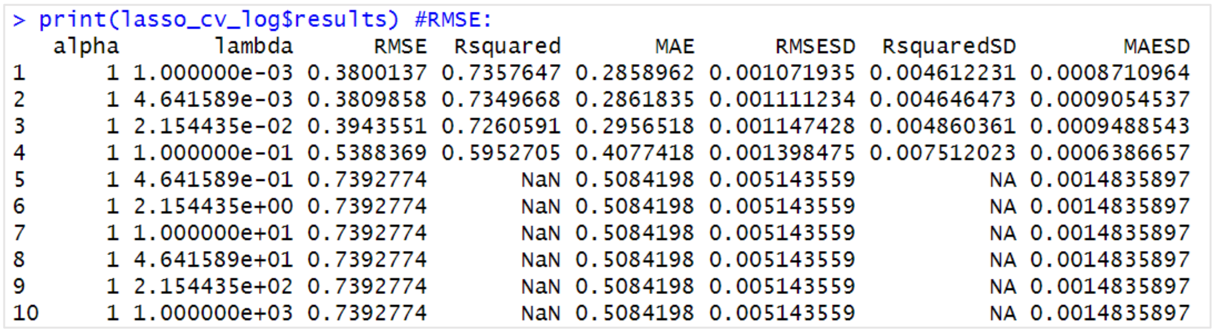
5.2.3.3 Lasso Regression Model After Log Transformation of Purchase Values

- Using the 5-fold cross-validation setup and the grid defined in Code Boxes 5.6 and 5.7, I trained a Lasso regression model on the log-transformed purchase values. The model performance was evaluated across different λ values, and the results including RMSE, R-squared, and MAE are shown in Code Box 5.15. The best RMSE achieved was approximately 0.3800.



```r
#Lasso_log
lasso_cv_log = train(log_purchase ~ . -purchase,
                     data = df_model_log,
                     method = "glmnet",
                     trControl = control_5,
                     tuneGrid = grid_lasso,
                     family = "gaussian")
print(lasso_cv_log$results) #RMSE: 0.3800137
```

Code Box 5.14: Training Lasso Regression with Logged Purchase



```
> print(lasso_cv_log$results) #RMSE:
   alpha       lambda      RMSE  Rsquared       MAE      RMSESD  RsquaredSD        MAESD
1      1 1.000000e-03 0.3800137 0.7357647 0.2858962 0.001071935 0.004612231 0.0008710964
2      1 4.641589e-03 0.3809858 0.7349668 0.2861835 0.001111234 0.004646473 0.0009054537
3      1 2.154435e-02 0.3943551 0.7260591 0.2956518 0.001147428 0.004860361 0.0009488543
4      1 1.000000e-01 0.5388369 0.5952705 0.4077418 0.001398475 0.007512023 0.0006386657
5      1 4.641589e-01 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
6      1 2.154435e+00 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
7      1 1.000000e+01 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
8      1 4.641589e+01 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
9      1 2.154435e+02 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
10     1 1.000000e+03 0.7392774       NaN 0.5084198 0.005143559          NA 0.0014835897
```

Code Box 5.15: Result of Lasso Regression with Logged Purchase

5.2.3.4 Current Findings with Lasso/Logged Lasso Regression

- In the Lasso regression using raw purchase values, the model achieved an RMSE of approximately 3014.173 and an R-squared of 0.639925. However, similar to Ridge regression on raw data, the large scale and skewed distribution of purchase values limited the regularization effect, and performance gains were marginal.

- After applying a log transformation to the purchase values, the Lasso regression model performed significantly better. The best RMSE dropped to approximately 0.3800, and the R-squared increased to 0.735764. This indicates that log transformation stabilized the variance and mitigated the effect of outliers, allowing the Lasso model to better leverage its variable selection capability.

- The results from Code Boxes 5.12 to 5.15 support this improvement, showing that Lasso is more effective when applied to a normalized target variable. Additionally, the model set several coefficients to zero, effectively identifying the most influential features in the dataset.

- Overall, Lasso regression with log-transformed purchase values demonstrated enhanced performance and interpretability, making it a valuable model for understanding key purchase drivers in Walmart's dataset.

Model	**RMSE**	**R-squared**	**MAE**
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	3027.535	0.638971	2287.476
Ridge Regression (Log)	0.3820	0.734997	0.285639
Lasso Regression (Raw)	3014.173	0.639929	2282.700
Lasso Regression (Log)	0.3800	0.737647	0.285896
Random Forest (Raw)	-	-	-
Random Forest (Log)	-	-	-
GBM (Raw)	-	-	-
GBM (Log-Transformed)	-	-	-
XGBoost (Raw)	-	-	-
XGBoost (Log-Transformed)	-	-	-
Neural Network (Log)	-	-	-

Table 5.4: Current Model Performance Summary (Lasso Regression Models)

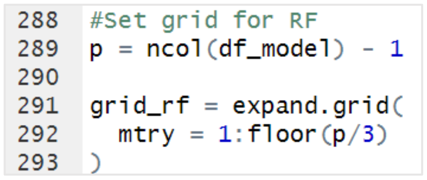
5.2.4 Random Forest

- Random Forest was selected for its ability to handle non-linear relationships and complex feature interactions, which are likely to exist in customer purchasing behavior data. Unlike linear models, Random Forest does not assume any specific functional form between features and the target variable, making it suitable for capturing hidden patterns in large, high-dimensional datasets like Walmart's.

- Additionally, Random Forest is robust to outliers and multicollinearity, and it provides built-in feature importance metrics that can help identify the most influential predictors. These strengths make it a strong candidate for modeling real-world retail data with diverse customer and product attributes.

5.2.4.1 Hyperparameter Utilization for Random Forest

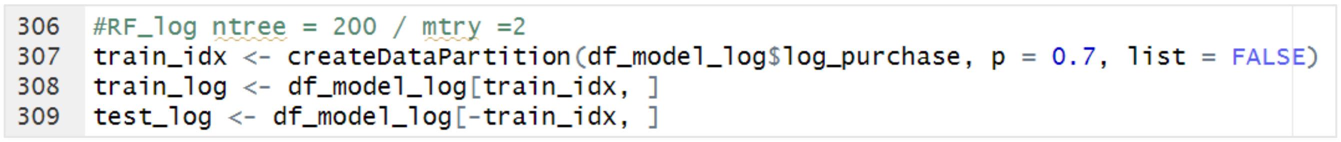
- To optimize the performance of the Random Forest model, two key hyperparameters were considered:
    - **mtry**: the number of features randomly selected at each split.
    - **ntree**: the number of trees to grow in the ensemble.
- To begin the tuning process, the number of predictors was stored in variable p, calculated as p = ncol(df_model) - 1. Then, mtry values were generated using expand.grid() from 1 to approximately one-third of the total features (*Code Box 5.15*)



```
#Set grid for RF
p = ncol(df_model) - 1

grid_rf = expand.grid(
  mtry = 1:floor(p/3)
)
```

Code Box 5.15: Grid Setup for mtry Tuning in Random Forest

- Additionally, although the original intention was to use 5-fold cross-validation, training time constraints made it necessary to split the dataset manually using createDataPartition(). This approach allowed for faster experimentation, especially when increasing ntree to values such as 200 or 500. (*Code Box 5.16*)



```
#RF_log ntree = 200 / mtry =2
train_idx <- createDataPartition(df_model_log$log_purchase, p = 0.7, list = FALSE)
train_log <- df_model_log[train_idx, ]
test_log <- df_model_log[-train_idx, ]
```

Code Box 5.16: Manual Train-Test Split Using createDataPartition()

5.2.4.2 Random Forest Process

- To explore the performance of the Random Forest model, four experimental setups were tested based on different combinations of data transformations, hyperparameters, and validation approaches:

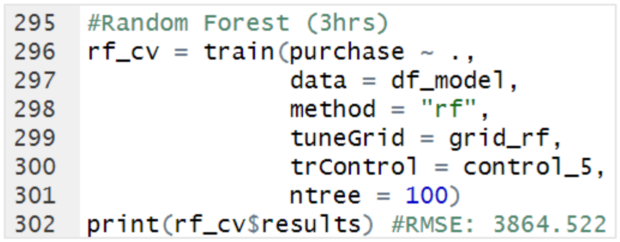
    1. CV-based Training with mtry Tuning (Standard Approach)
    - Dataset: df_model
    - Method: 5-fold Cross-Validation (control_5) with grid search on mtry
    - ntree: 100
    - Result: RMSE ≈ 3864.522

    2. Logged Data + Manual Split + ntree = 200
    - Dataset: df_model_log
    - Method: Manual split using createDataPartition() (no CV)
    - mtry: Fixed at 2
    - ntree: 200
    - Result: RMSE ≈ 10531.722 (after back-transforming from log scale)

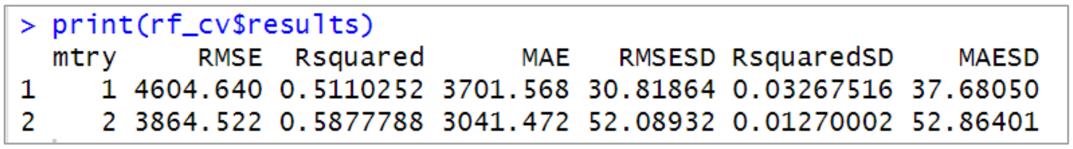
    3. Logged Data + Manual Split + ntree = 500
    - Same as above, with ntree increased to 500
    - Result: RMSE ≈ 10096.23

    4. Raw Data + Manual Split + ntree = 300
    - Dataset: df_model
    - Method: Manual split using createDataPartition() (no CV)
    - mtry: Fixed at 2
    - ntree: 300
    - Result: ≈ 3878.722

- As illustrated in *Code Boxes 5.17 to 5.20*, different combinations of data transformation (log vs. raw), validation method (cross-validation vs. manual partitioning), and hyperparameter settings (mtry, ntree) were applied to train and evaluate the Random Forest models. These examples demonstrate how the presence or absence of cross-validation, as well as whether the target variable is log-transformed, can influence both the model structure and resulting performance.

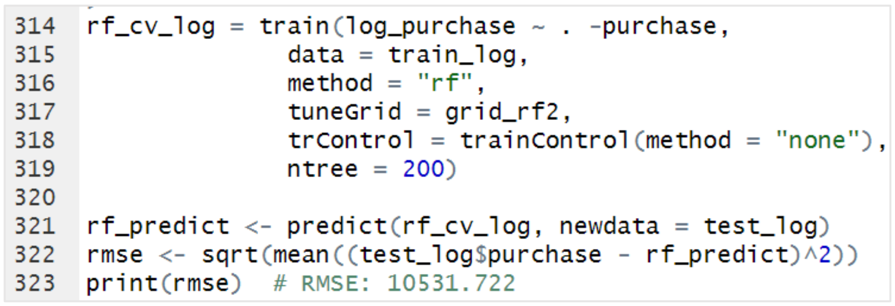


```r
#Random Forest (3hrs)
rf_cv = train(purchase ~ .,
              data = df_model,
              method = "rf",
              tuneGrid = grid_rf,
              trControl = control_5,
              ntree = 100)
print(rf_cv$results) #RMSE: 3864.522
```

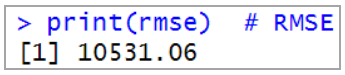


```
> print(rf_cv$results)
  mtry     RMSE  Rsquared      MAE   RMSESD RsquaredSD    MAESD
1    1 4604.640 0.5110252 3701.568 30.81864 0.03267516 37.68050
2    2 3864.522 0.5877788 3041.472 52.08932 0.01270002 52.86401
```

Code Box 5.17: Random Forest with Cross-Validation and mtry Tuning (ntree = 100)

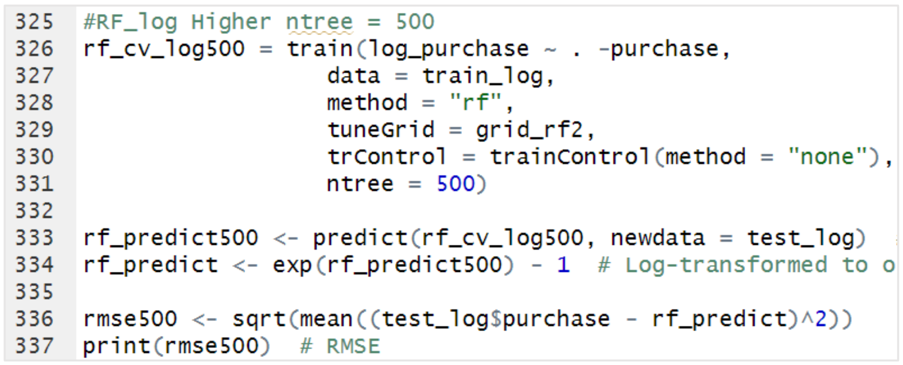


```r
rf_cv_log = train(log_purchase ~ . -purchase,
              data = train_log,
              method = "rf",
              tuneGrid = grid_rf2,
              trControl = trainControl(method = "none"),
              ntree = 200)

rf_predict <- predict(rf_cv_log, newdata = test_log)
rmse <- sqrt(mean((test_log$purchase - rf_predict)^2))
print(rmse)  # RMSE: 10531.722
```

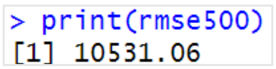


```
> print(rmse)  # RMSE
[1] 10531.06
```

Code Box 5.18: Random Forest on Log-Transformed Data without CV (ntree = 200)

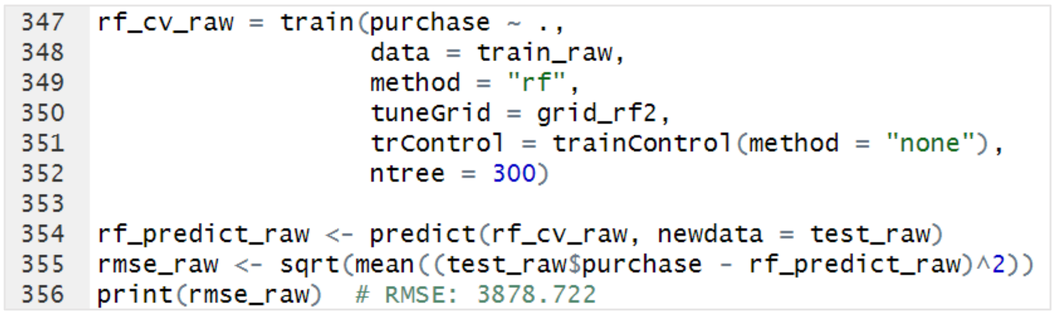


```r
#RF_log Higher ntree = 500
rf_cv_log500 = train(log_purchase ~ . -purchase,
                     data = train_log,
                     method = "rf",
                     tuneGrid = grid_rf2,
                     trControl = trainControl(method = "none"),
                     ntree = 500)

rf_predict500 <- predict(rf_cv_log500, newdata = test_log)
rf_predict <- exp(rf_predict500) - 1  # Log-transformed to o

rmse500 <- sqrt(mean((test_log$purchase - rf_predict)^2))
print(rmse500)  # RMSE
```



```
> print(rmse500)
[1] 10531.06
```

Code Box 5.19: Random Forest on Log-Transformed Data with Higher ntree (ntree = 500)



```r
rf_cv_raw = train(purchase ~ .,
                  data = train_raw,
                  method = "rf",
                  tuneGrid = grid_rf2,
                  trControl = trainControl(method = "none"),
                  ntree = 300)

rf_predict_raw <- predict(rf_cv_raw, newdata = test_raw)
rmse_raw <- sqrt(mean((test_raw$purchase - rf_predict_raw)^2))
print(rmse_raw)  # RMSE: 3878.722
```

Code Box 5.20: Random Forest on Raw Data without CV (ntree = 300)

5.2.4.3 Current Findings with Random Forest

- Among the four experimental setups, the model trained using 5-fold cross-validation with mtry tuning (*Code Box 5.17*) achieved an RMSE of approximately 3864.522. Despite being the most methodologically robust, this model showed relatively higher error, possibly due to the limited number of trees (ntree = 100) and high variance across the folds.

- When applying log transformation and training with ntree = 200 without cross-validation (*Code Box 5.18*), the RMSE increased significantly to 10531.06 after back-transforming the predictions. This deterioration can be attributed to the amplification effect caused by the inverse log transformation (exponential), which can exaggerate prediction errors.

- Even when increasing ntree to 500 with the same log-transformed setup (*Code Box 5.19*), the RMSE remained similar at 10531.06, indicating that increasing the number of trees did not help in reducing error under this configuration.

- On the other hand, the model trained on raw data with ntree = 300 and no cross-validation (Code Box 5.20) resulted in an RMSE of *3878.722*, which is comparable to the CV-based setup. This suggests that the model performed reasonably well without log transformation and that over-regularization or transformation instability may have hurt the log-based versions.

Model	**RMSE**	**R-squared**	**MAE**
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	3027.535	0.638971	2287.476
Ridge Regression (Log)	0.3820	0.734997	0.285639
Lasso Regression (Raw)	3014.173	0.639929	2282.700
Lasso Regression (Log)	0.3800	0.737647	0.285896
Random Forest (Raw)	3864.522	-	-
Random Forest (Log)	10531.06	-	-
GBM (Raw)	-	-	-
GBM (Log-Transformed)	-	-	-
Neural Network (Log)	-	-	-

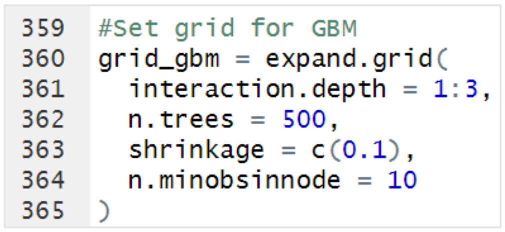
Table 5.5: Current Model Performance Summary (Random Forest Models)

5.2.5 Gradient Boosting Machine (GBM)

- Gradient Boosting Machine (GBM) is an ensemble method that builds models sequentially, with each new tree attempting to correct the errors of the previous ones.
- Unlike Random Forest, which builds trees independently, GBM captures complex non-linear relationships more precisely by focusing on residual errors.
- However, it is more prone to overfitting, so careful hyperparameter tuning is essential to ensure generalization.

5.2.5.1 Hyperparameter Utilization for GBM

- To optimize the GBM model, several key hyperparameters were tuned:
    - interaction.depth: Controls the maximum depth of individual trees (values: 1 to 3).
    - n.trees: The number of boosting iterations (trees). A higher value improves learning capacity but increases overfitting risk (value: 500).
    - shrinkage: The learning rate; a smaller value slows down learning but helps improve generalization (value: 0.1).
    - n.minobsinnode: The minimum number of observations in a node required to split. Larger values reduce model complexity and help prevent overfitting (value: 10).
- These parameters were combined using expand.grid() to define a tuning grid (*Code Box 5.21*)



```r
#Set grid for GBM
grid_gbm = expand.grid(
  interaction.depth = 1:3,
  n.trees = 500,
  shrinkage = c(0.1),
  n.minobsinnode = 10
)
```

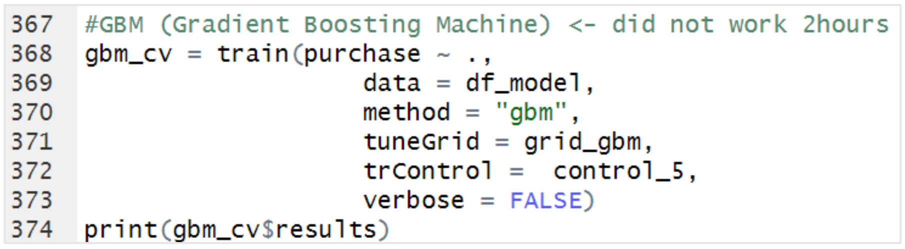
Code Box 5.21: GBM Grid Setup

5.2.5.2 Gradient Boosting Machine Process

- Initially, I attempted to train the GBM model using 5-fold cross-validation and a wide parameter grid (Code Box 5.22). However, this approach did not complete even after 2 hours, likely due to the computational intensity of tuning multiple hyperparameters over many folds. (*Code Box 5.22*)

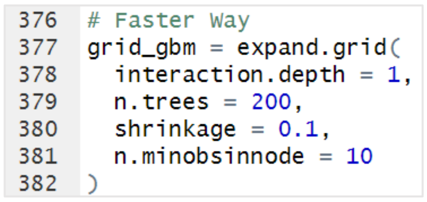
- To improve efficiency, I simplified the grid (Code Box 5.23)
- Additionally, I disabled cross-validation and instead used manual data splitting via createDataPartition() to train the model directly on a training subset and evaluate on a hold-out test set.

- Both log-transformed and raw purchase versions of the GBM were trained with this simplified setup. Their RMSEs were calculated after prediction (and inverse-log transformation where applicable).



```r
#GBM (Gradient Boosting Machine) <- did not work 2hours
gbm_cv = train(purchase ~ .,
                   data = df_model,
                   method = "gbm",
                   tuneGrid = grid_gbm,
                   trControl =  control_5,
                   verbose = FALSE)
print(gbm_cv$results)
```

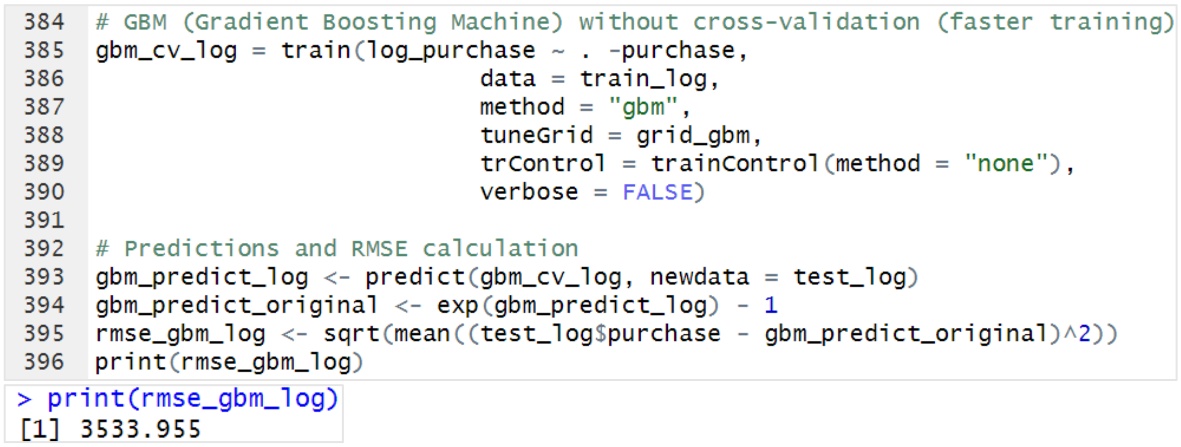
Code Box 5.22: Initial GBM Training Attempt with Full Grid and Cross-Validation (Did Not Complete)



```r
# Faster Way
grid_gbm = expand.grid(
  interaction.depth = 1,
  n.trees = 200,
  shrinkage = 0.1,
  n.minobsinnode = 10
)
```

Code Box 5.23: Simplified GBM Grid Setup for Faster Training

- Using the simplified grid setup without cross-validation (as shown in *Code Box 5.23 and 5.24)*, the GBM model trained on log-transformed purchase values achieved an RMSE of approximately 3533.955 after back-transforming the predictions. This result demonstrates a significant improvement in training efficiency while still yielding a reasonable performance.

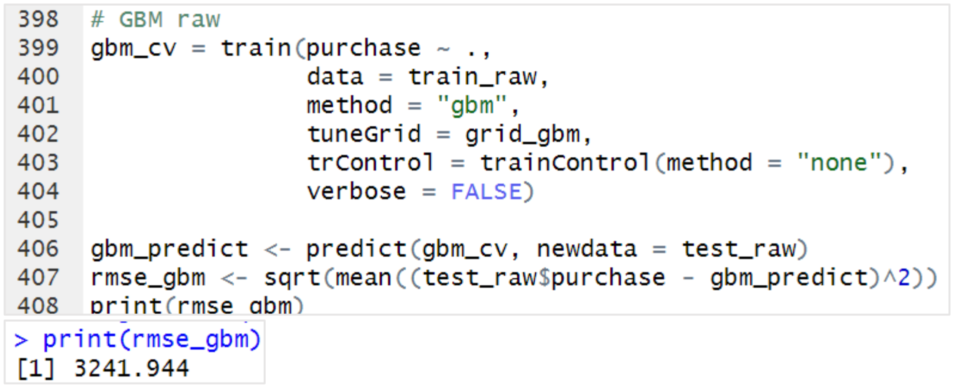


```r
# GBM (Gradient Boosting Machine) without cross-validation (faster training)
gbm_cv_log = train(log_purchase ~ . -purchase,
                                data = train_log,
                                method = "gbm",
                                tuneGrid = grid_gbm,
                                trControl = trainControl(method = "none"),
                                verbose = FALSE)

# Predictions and RMSE calculation
gbm_predict_log <- predict(gbm_cv_log, newdata = test_log)
gbm_predict_original <- exp(gbm_predict_log) - 1
rmse_gbm_log <- sqrt(mean((test_log$purchase - gbm_predict_original)^2))
print(rmse_gbm_log)
```

```
> print(rmse_gbm_log)
[1] 3533.955
```

Code Box 5.24: Log-Transformed GBM Model without Cross-Validation (RMSE = 3533.955)

- Additionally, using the same simplified grid setup, I also trained a GBM model on the raw purchase values (*Code Box 5.25*). This approach achieved a lower RMSE of approximately 3241.944, indicating that the model performed slightly better without log transformation in this case. This may suggest that the amplification effect from back-transforming log predictions introduced additional error in the log-transformed version.

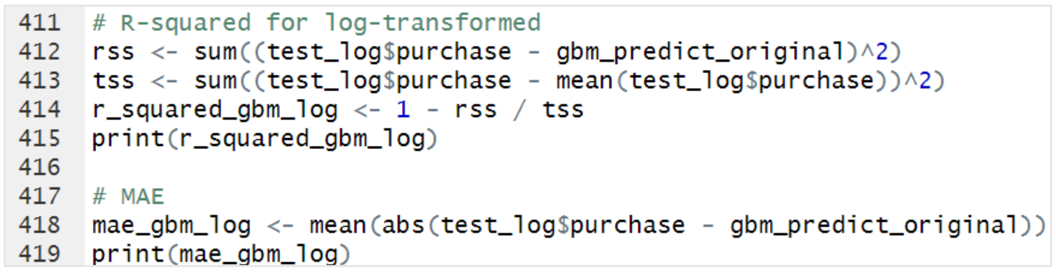


```r
# GBM raw
gbm_cv = train(purchase ~ .,
                 data = train_raw,
                 method = "gbm",
                 tuneGrid = grid_gbm,
                 trControl = trainControl(method = "none"),
                 verbose = FALSE)

gbm_predict <- predict(gbm_cv, newdata = test_raw)
rmse_gbm <- sqrt(mean((test_raw$purchase - gbm_predict)^2))
print(rmse_gbm)
```

```
> print(rmse_gbm)
[1] 3241.944
```

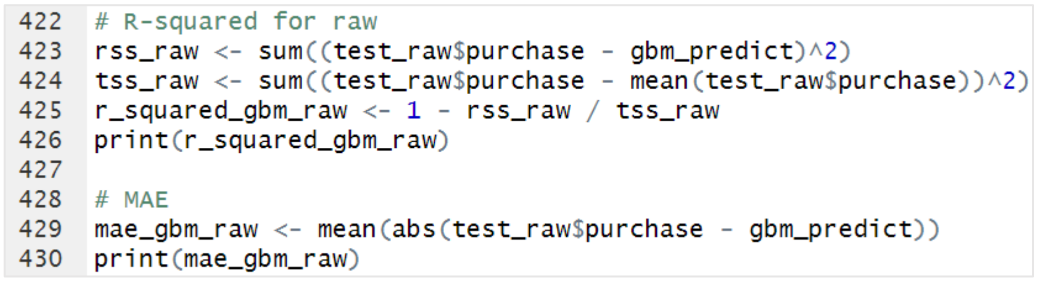
Code Box 5.25: Raw GBM Model without Cross-Validation (RMSE = 3241.944)

- Additionally, I computed the R-squared and MAE values for both versions of the GBM model.
  - For the log-transformed GBM, the model achieved an R-squared of 0.506 and MAE of 2607.78.
  - For the raw GBM, the model performed slightly better with an R-squared of 0.583 and MAE of 2454.093.
  - These results suggest that, in this case, skipping the log transformation yielded marginally better predictive accuracy and interpretability.



```r
# R-squared for log-transformed
rss <- sum((test_log$purchase - gbm_predict_original)^2)
tss <- sum((test_log$purchase - mean(test_log$purchase))^2)
r_squared_gbm_log <- 1 - rss / tss
print(r_squared_gbm_log)

# MAE
mae_gbm_log <- mean(abs(test_log$purchase - gbm_predict_original))
print(mae_gbm_log)
```

Code Box 5.26: Evaluation Metrics for Log-Transformed GBM Model (R² and MAE)



```
# R-squared for raw
rss_raw <- sum((test_raw$purchase - gbm_predict)^2)
tss_raw <- sum((test_raw$purchase - mean(test_raw$purchase))^2)
r_squared_gbm_raw <- 1 - rss_raw / tss_raw
print(r_squared_gbm_raw)

# MAE
mae_gbm_raw <- mean(abs(test_raw$purchase - gbm_predict))
print(mae_gbm_raw)
```

Code Box 5.27: Evaluation Metrics for Raw GBM Model (R² and MAE)

5.2.5.3 Current Findings with GBM

- When comparing the two variations of the GBM model, the version using raw data outperformed the log-transformed counterpart.
- The GBM (Raw) model achieved an RMSE of 3241.944, R-squared of 0.582770, and MAE of 2454.093, showing lower prediction error than the log-transformed version (RMSE: 3533.955). This suggests that the inverse log transformation (via exp) may have amplified prediction errors, and that the GBM model can perform effectively even without log transformation.

Model	**RMSE**	**R-squared**	**MAE**
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	3027.535	0.638971	2287.476
Ridge Regression (Log)	0.3820	0.734997	0.285639
Lasso Regression (Raw)	3014.173	0.639929	2282.700
Lasso Regression (Log)	0.3800	0.737647	0.285896
Random Forest (Raw)	3864.522	-	-
Random Forest (Log)	10531.06	-	-
GBM (Log-Transformed)	3533.955	0.505980	2607.780
GBM (Raw)	3241.944	0.582770	2454.093
Neural Network (Log)	-	-	-

Table 5.6: Current Model Performance Summary (GBM Models)

5.2.6 XGBoost

- XGBoost is an advanced implementation of gradient boosting that offers optimized speed and performance through techniques such as regularization, parallelization, and tree pruning. It is known for delivering high predictive accuracy and has become a popular choice in many machine learning competitions and real-world applications.

- **Why It Was Not Used in This Study**: Despite its advantages, XGBoost was not included in the modeling process for this project. Since GBM (Gradient Boosting Machine), a similar boosting-based algorithm, had already been evaluated and showed lower performance compared to simpler linear models, based on RMSE, R-squared, and MAE metrics,the decision was made to omit XGBoost.

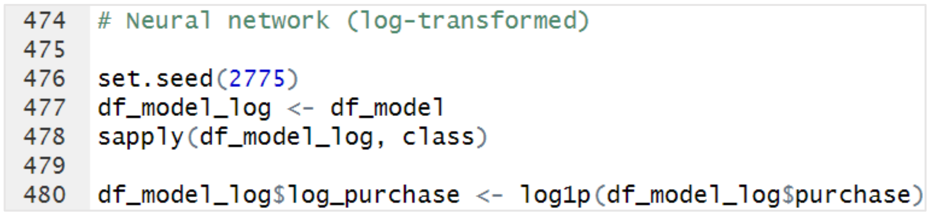
- The goal was to prioritize model interpretability and efficiency. Given the significantly longer training times and the diminishing returns expected from a similar boosting method, XGBoost was excluded to maintain focus on more impactful model comparisons.

5.2.7 Neural Network

- To explore non-linear relationships in purchase prediction, a neural network model was implemented using log-transformed purchase values. As shown in Code Boxes 5.28 to 5.31, factor variables were re-encoded via one-hot encoding, and data was normalized to ensure stable training. A random 20% sample of the dataset was used to reduce computational burden, followed by a 70:30 train-test split.

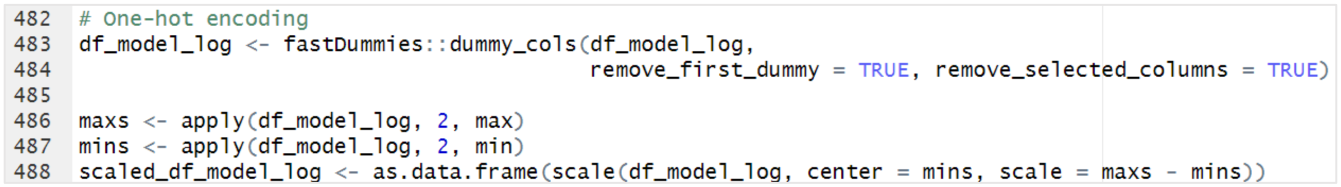
- However, during training, the model exceeded 5 hours of runtime, leading to the early termination of the process. Given the relatively high training time and the already strong performance of simpler linear models (especially the log-transformed linear regression with RMSE of 0.3799), the decision was made not to proceed further with the neural network model.

- Thus, neural networks were excluded from the final evaluation due to computational inefficiency and diminishing performance gains relative to simpler, more interpretable models.



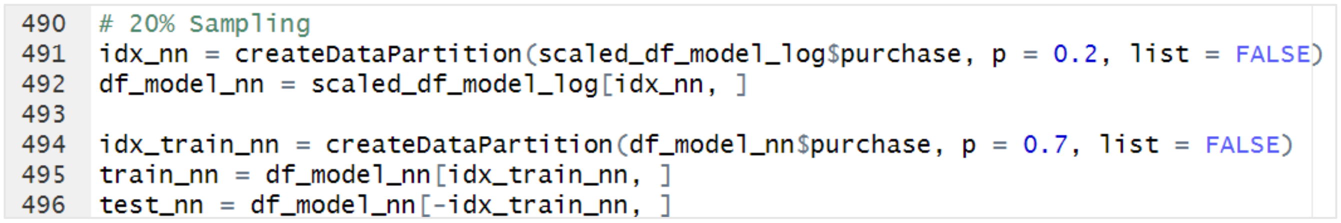
```r
# Neural network (log-transformed)

set.seed(2775)
df_model_log <- df_model
sapply(df_model_log, class)

df_model_log$log_purchase <- log1p(df_model_log$purchase)
```

Code Box 5.28: Log Transformation for Neural Network Input



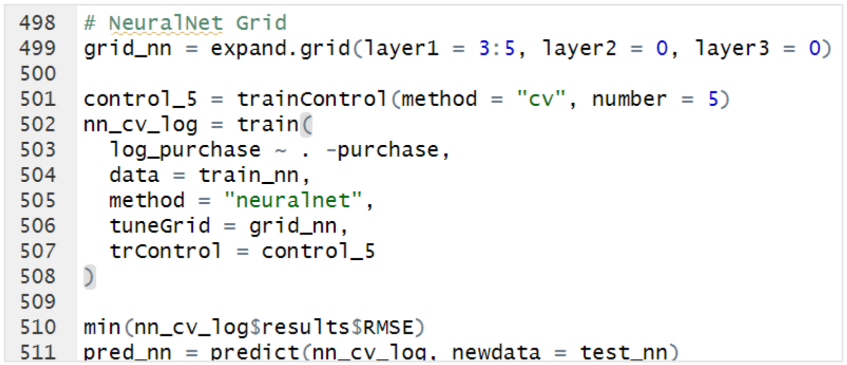
```r
# One-hot encoding
df_model_log <- fastDummies::dummy_cols(df_model_log,
                                        remove_first_dummy = TRUE, remove_selected_columns = TRUE)

maxs <- apply(df_model_log, 2, max)
mins <- apply(df_model_log, 2, min)
scaled_df_model_log <- as.data.frame(scale(df_model_log, center = mins, scale = maxs - mins))
```

Code Box 5.29: One-Hot Encoding and Feature Scaling



```r
# 20% Sampling
idx_nn = createDataPartition(scaled_df_model_log$purchase, p = 0.2, list = FALSE)
df_model_nn = scaled_df_model_log[idx_nn, ]

idx_train_nn = createDataPartition(df_model_nn$purchase, p = 0.7, list = FALSE)
train_nn = df_model_nn[idx_train_nn, ]
test_nn = df_model_nn[-idx_train_nn, ]
```

Code Box 5.30: Data Sampling and Train-Test Split for Neural Network



```r
# NeuralNet Grid
grid_nn = expand.grid(layer1 = 3:5, layer2 = 0, layer3 = 0)

control_5 = trainControl(method = "cv", number = 5)
nn_cv_log = train(
  log_purchase ~ . -purchase,
  data = train_nn,
  method = "neuralnet",
  tuneGrid = grid_nn,
  trControl = control_5
)

min(nn_cv_log$results$RMSE)
pred_nn = predict(nn_cv_log, newdata = test_nn)
```

Code Box 5.31: Neural Network Training with Grid Search and Cross-Validation

5.3 Final Model Performance Summary

- Among all the models evaluated, the log-transformed Linear Regression model demonstrated the best overall performance.

- This conclusion is based on two key dimensions:
    1. performance metrics
    2. practical business usage.

    - **From a performance perspective**, the log-transformed linear regression achieved the lowest RMSE (0.3799), the highest R-squared (0.7359), and the lowest MAE (0.2859) among all models tested. These results indicate strong predictive power with minimal errors and a well-fitted model.
    - **From a practical business usage standpoint**, linear regression is also the most interpretable and computationally efficient model. It provides clear insights into the impact of each predictor on purchase amount, which can support business decisions such as customer segmentation, marketing strategy, and demand forecasting.

- The consistently strong performance of log-transformed Linear, Ridge, and Lasso models also indicates that a linear relationship exists between features and purchase amount, especially when the target variable is normalized via log transformation.

- In contrast, non-linear models such as Random Forest, GBM, and Neural Network not only required significantly longer training time but also showed inferior results in terms of RMSE and R-squared. This suggests that the added complexity of non-linear methods did not translate into better performance for this dataset.

6 Recommendations & Conclusion

6.1 Key Findings

- Upon comparing both raw and log-transformed versions of each model, it was evident that log-transformed models consistently outperformed raw models in terms of RMSE, R-squared, and MAE, especially in linear approaches. (*Table 6.1*)

- Among all models, log-transformed Linear Regression achieved the best overall performance with the lowest RMSE (0.3799), highest R-squared (0.7359), and lowest MAE (0.2858), followed closely by Lasso Regression (Log) and Ridge Regression (Log).

- These findings suggest that Walmart's purchase data follows a linear trend, and applying log transformation stabilized the variance and improved predictive power by reducing the impact of outliers. Therefore, the best performing model was **Linear Regression with log transformation and 5-fold cross-validation**, based on its balance between **accuracy**, **simplicity**, and **training time efficiency**. Other strong performers include Ridge (Log) and Lasso (Log) models.

Model	**RMSE**	**R-squared**	**MAE**
Linear Regression (Raw)	3014.089	0.639946	2282.591
Linear Regression (Log)	0.3799	0.735886	0.285868
Ridge Regression (Raw)	3027.535	0.638971	2287.476
Ridge Regression (Log)	0.3820	0.734997	0.285639
Lasso Regression (Raw)	3014.173	0.639929	2282.700
Lasso Regression (Log)	0.3800	0.737647	0.285896
Random Forest (Raw)	3864.522	-	-
Random Forest (Log)	10531.06	-	-
GBM (Raw)	3533.955	0.505980	2607.780
GBM (Log-Transformed)	3241.944	0.582770	2454.093
Neural Network (Log)	-	-	-

Table 6.1: Final Model Performance Summary

6.2 Limitations

- ***Computational Limitations***: Due to hardware constraints, models like Neural Network and full-grid GBM with cross-validation could not be completed within a reasonable time frame (e.g., >5 hours).
- ***Sampling Constraints***: Cross-validation was not feasible for some models due to long training times, leading to reliance on manual train/test splits.

6.3 Recommendation:

- For business applications, start with log-transformed Linear Regression to predict purchase values efficiently and accurately.
- Use the log-transformed model output to help identify high-spending users, allowing for premium targeting and customer segmentation strategies.
- If needed, supplement predictions with Ridge or Lasso regression for better feature selection or regularization, especially in higher-dimensional settings.

6.4 Future Work

- There are several potential directions to enhance this analysis in future iterations.
    - ***Feature Engineering***: Incorporate temporal, regional, or behavioral features to enhance predictive power.
    - ***Model Expansion***: Revisit complex models like XGBoost or Neural Networks in environments with sufficient computational resources.
    - ***Business Integration***: Investigate how the best model can be used to recommend premium products or identify potential high-spending users.

6.5 Conclusion

- This project evaluated seven machine learning models to predict customer purchase behavior. Early exploration revealed significant outliers and skewed distributions, which were effectively addressed through log transformation. Linear models, particularly those using log-transformed targets, demonstrated superior performance and interpretability.

- Ultimately, this analysis highlights the importance of examining data structure early in the modeling process. Gaining insights from exploratory analysis was key to selecting the best model configuration and improving predictive results, , such as identifying outliers and distribution characteristics.