# Predict Used Car Selling Price

By

**Srivathsan Raveendran**

**Machine Learning for Big Data**

**Frankfurt School of Finance and Management**

# Acknowledgments

We would like to extend our heartfelt thanks and deepest appreciation to Prof. Dr. Peter Roßbach for his unwavering support, insightful guidance, and invaluable mentorship throughout the entire duration of this project. His expertise and encouragement were pivotal in shaping the direction and success of our work, and we are profoundly grateful for his dedication to fostering a culture of learning and innovation.

Additionally, we would like to express our gratitude to the entire cohort of students from the "Machine Learning for Big Data" course. Their collaborative spirit, thought-provoking discussions, and shared knowledge enriched our understanding and inspired us to push the boundaries of our capabilities. This project stands as a testament to the collective efforts and contributions of everyone involved, and we thank all who played a part in this incredible journey.

# LITERATURE REVIEW

Several studies have explored predicting used car prices across the globe using diverse datasets, methodologies, and machine learning models, achieving varying degrees of accuracy ranging from 50% to 98%. For instance, Noor and Jan (2017) conducted research using data collected from Pakistan's Pak Wheels website, containing 1,699 records post-processing. By employing multiple linear regression and applying a variable selection technique to reduce model complexity, they achieved an exceptional accuracy of 98%. Similarly, Kuiper (2008) utilized data from General Motors' 2005 production to implement multivariate regression after narrowing the dataset to relevant variables, demonstrating that selecting key attributes enhances prediction efficiency.

In contrast, Monburinon et al. (2018) investigated a dataset from a German e-commerce platform, comprising 304,133 records and 11 attributes. Their study compared several machine learning models, with the Gradient Boosted Regression Tree achieving the lowest Mean Absolute Error (MAE) of 0.28. The authors emphasized the importance of adjusting parameters and suggested one-hot encoding for categorical variables to improve future model performance. Meanwhile, Pudaruth (2014) examined car prices in Mauritius using models such as Decision Trees, K-Nearest Neighbors (KNN), Naïve Bayes, and Multiple Regression, achieving accuracy rates of 60-70%. He noted the need for larger datasets and more advanced techniques due to the limitations of discretizing data in Decision Tree and Naïve Bayes models, which led to inaccuracies.

K. Samruddhi and Kumar (2020) approached the problem using KNN on Kaggle's dataset, containing 14 attributes. Through fine-tuning the value of k and adjusting the train-test split ratios, they achieved an accuracy of 85%. Their model also benefited from cross-validation using 5- and 10-fold splits, highlighting the importance of robust testing.

Further highlighting the potential of neural networks, Gongqi et al. (2011) developed a hybrid model combining Backpropagation Neural Networks and nonlinear curve fitting to predict car prices accurately. Similarly, Jian Da Wu (2017) proposed an Adaptive Neuro-Fuzzy Inference System (ANFIS) that integrated fuzzy logic and ANN capabilities. This innovative approach outperformed traditional ANN models, providing superior prediction accuracy and user-friendly outputs through a graphical interface.

Listiani (2009) demonstrated the advantages of SVM over multiple linear regression in predicting leased car prices, particularly for high-dimensional datasets. While SVM achieved greater accuracy, its computational complexity significantly outweighed the simplicity and speed of linear regression, making the latter more appropriate for small-scale studies. This aligns with findings from Pudaruth (2014) and Kuiper (2008), who favored simpler regression-based approaches for smaller or low-dimensional datasets.

Together, these studies illustrate the evolution of predictive methodologies for used car prices, emphasizing the importance of dataset quality, feature selection, and model complexity in achieving reliable results. Each approach—whether regression-based or leveraging advanced neural networks—offers unique insights and trade-offs, paving the way for more refined applications in the future.

# 1. Introduction

## 1.1Scenario

For a used car dealer, having precise and reliable information about the attainable price of a used car is crucial for making informed purchasing decisions. This knowledge enables the dealer to determine an optimal purchase price, ensuring that a desirable profit margin is maintained while remaining competitive in the market. Recognizing the importance of accurate price predictions, this project aims to develop a robust predictive model for used car prices. By leveraging data analytics and machine learning techniques, the model will provide actionable insights that enhance decision-making and profitability in the dynamic used car market.

## 1.2 Project goal

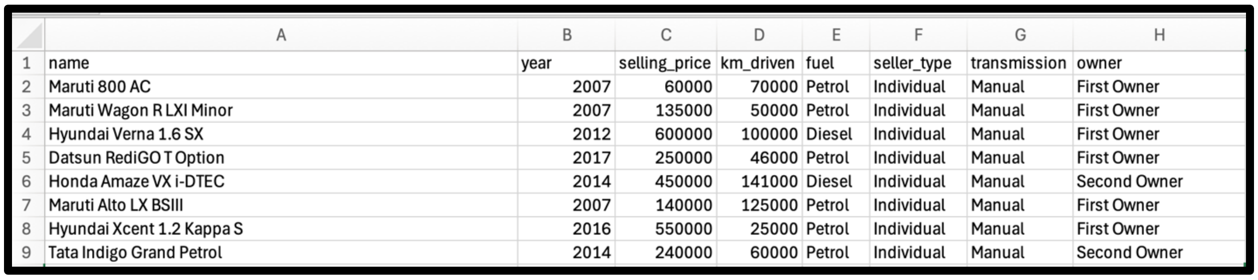
This project aims to develop a predictive model for used car prices to assist dealers in making informed purchase decisions and achieving their desired profit margins. By accurately estimating the attainable selling price of a used car, dealers can optimize their procurement strategies, ensuring fair pricing and sustainable business operations. These models are designed to provide valuable insights into the used car market, helping stakeholders navigate price variability and market trends effectively.

## 1.3 Exploring the raw data

This dataset contains information on 4,340 used cars, including their specifications, sales prices, and ownership details.

Columns and Descriptions:

| Sr# | Column Name | Datatype | Description |
|---|---|---|---|
| **1** | name | Object | The make and model of the car |
| **2** | year | Int | The year the car was manufactured (ranging from 1992 to 2020) |
| **3** | selling_price | Int | The price at which the car was sold (ranging from 20,000 to 8,900,000) |
| **4** | km_driven | Int | The total kilometres driven by the car (ranging from 1 to 806,599 km) |
| **5** | fuel | Object | The type of fuel the car uses (e.g., Petrol, Diesel) |
| **6** | seller_type | Object | The type of seller (e.g., Individual, Dealer, or Trustmark Dealer) |
| **7** | transmission | Object | The transmission type of the car (Manual or Automatic) |
| **8** | owner | Object | Ownership status (First Owner, Second Owner, etc.) |



|  | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | name | year | selling_price | km_driven | fuel | seller_type | transmission | owner |
| 2 | Maruti 800 AC | 2007 | 60000 | 70000 | Petrol | Individual | Manual | First Owner |
| 3 | Maruti Wagon R LXI Minor | 2007 | 135000 | 50000 | Petrol | Individual | Manual | First Owner |
| 4 | Hyundai Verna 1.6 SX | 2012 | 600000 | 100000 | Diesel | Individual | Manual | First Owner |
| 5 | Datsun RediGO T Option | 2017 | 250000 | 46000 | Petrol | Individual | Manual | First Owner |
| 6 | Honda Amaze VX i-DTEC | 2014 | 450000 | 141000 | Diesel | Individual | Manual | Second Owner |
| 7 | Maruti Alto LX BSIII | 2007 | 140000 | 125000 | Petrol | Individual | Manual | First Owner |
| 8 | Hyundai Xcent 1.2 Kappa S | 2016 | 550000 | 25000 | Petrol | Individual | Manual | First Owner |
| 9 | Tata Indigo Grand Petrol | 2014 | 240000 | 60000 | Petrol | Individual | Manual | Second Owner |

## 1.4 Packages installed

library(dplyr) - Streamlines data manipulation with functions for filtering, selecting, and summarizing data.

library(tidyverse) -

library(ggplot2) - A powerful tool for creating elegant and customizable data visualizations

library(lubridate) - Simplifies working with dates and times, including parsing and calculations

library(stringr) - Makes string manipulation intuitive with functions for pattern matching and modification.

library(rpart)

library(rpart.plot)

library(ipred) - Implements bagging and other ensemble methods for improved predictive accuracy.

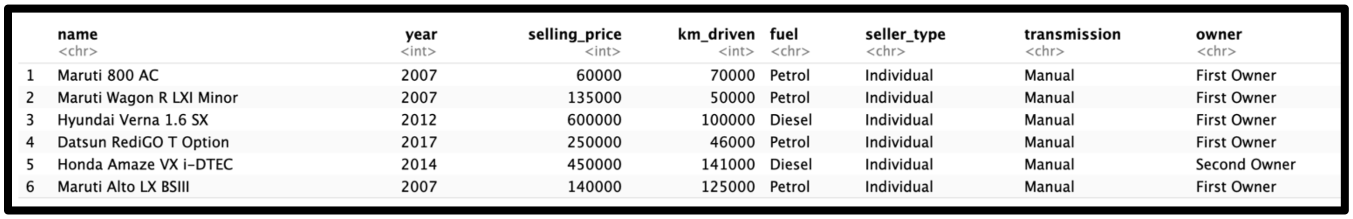
library(caret) - A comprehensive package for training and evaluating machine learning models

library(GGally) - Extends ggplot2 with functions like pair plots for visualizing variable relationships.

## 1.5 Renaming Columns:

The purpose of renaming columns in a dataset is to improve clarity, consistency, and usability during data analysis and modelling.

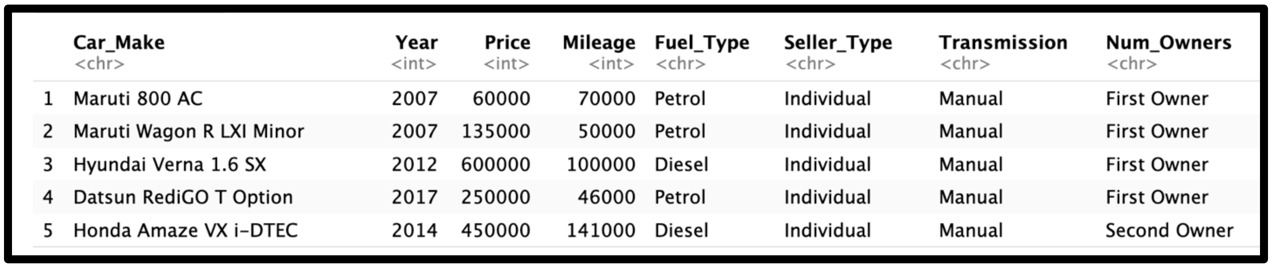
Before renaming:



```
  name                       year  selling_price  km_driven  fuel    seller_type  transmission  owner
  <chr>                     <int>          <int>      <int>  <chr>   <chr>        <chr>         <chr>
1 Maruti 800 AC              2007          60000      70000  Petrol  Individual   Manual        First Owner
2 Maruti Wagon R LXI Minor   2007         135000      50000  Petrol  Individual   Manual        First Owner
3 Hyundai Verna 1.6 SX       2012         600000     100000  Diesel  Individual   Manual        First Owner
4 Datsun RediGO T Option     2017         250000      46000  Petrol  Individual   Manual        First Owner
5 Honda Amaze VX i-DTEC      2014         450000     141000  Diesel  Individual   Manual        Second Owner
6 Maruti Alto LX BSIII       2007         140000     125000  Petrol  Individual   Manual        First Owner
```

| Original name | New name | Reason |
|---|---|---|
| name | Car_Make | The column stores car brand or model names, so Car_Make is more descriptive |
| year | Year | Capitalized for consistency |
| selling_price | Price | Simplified while retaining meaning |
| km_driven | Mileage | The new name reflects the meaning (distance travelled by the car) |
| fuel | Fuel_Type | Adds clarity about what the "fuel" column represents |
| seller_type | Seller_type | Capitalized for consistency |
| transmission | Transmission | No change, but kept consistent in style |
| owner | Num_Owners | Makes it clear that the column represents the number of previous owners |

After renaming:



| | Car_Make <chr> | Year <int> | Price <int> | Mileage <int> | Fuel_Type <chr> | Seller_Type <chr> | Transmission <chr> | Num_Owners <chr> |
|---|---|---|---|---|---|---|---|---|
| 1 | Maruti 800 AC | 2007 | 60000 | 70000 | Petrol | Individual | Manual | First Owner |
| 2 | Maruti Wagon R LXI Minor | 2007 | 135000 | 50000 | Petrol | Individual | Manual | First Owner |
| 3 | Hyundai Verna 1.6 SX | 2012 | 600000 | 100000 | Diesel | Individual | Manual | First Owner |
| 4 | Datsun RediGO T Option | 2017 | 250000 | 46000 | Petrol | Individual | Manual | First Owner |
| 5 | Honda Amaze VX i-DTEC | 2014 | 450000 | 141000 | Diesel | Individual | Manual | Second Owner |

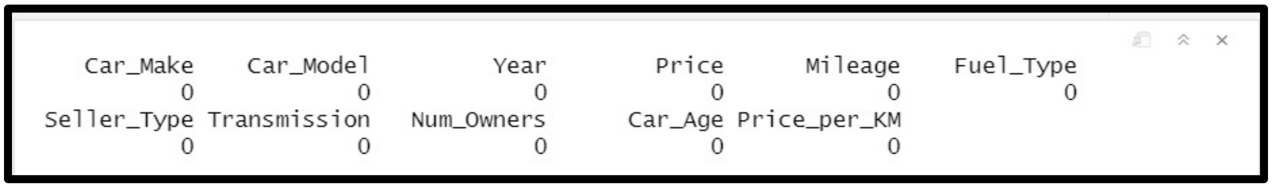
# 2. Data Preparation

## 2.1 Handling missing values

In tabular datasets, it is common to encounter incomplete records where certain fields are missing. To ensure data consistency, we must validate that each record has the expected number of fields and that each field contains the appropriate data type. When handling missing fields, we typically have two options: either discard the entire record or fill in the missing values using the average or median. In R, missing values are denoted as NA (Not Available), while invalid values, such as the result of division by zero, are represented as NaN (Not a Number).

***Code:***

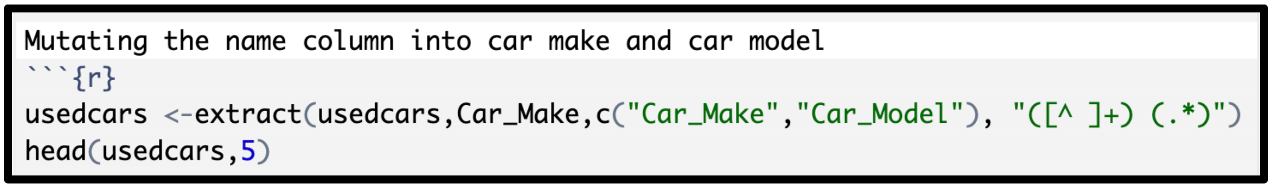
colSums(is.na(usedcars))



```
    Car_Make    Car_Model         Year        Price      Mileage    Fuel_Type 
           0            0            0            0            0            0 
 Seller_Type Transmission   Num_Owners      Car_Age Price_per_KM 
           0            0            0            0            0 
```

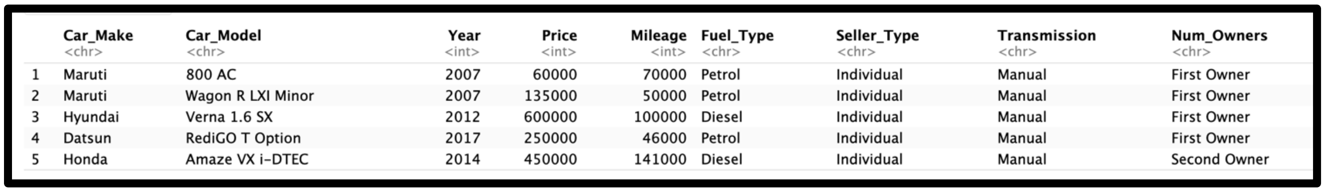
## 2.2 Splitting Car make and Car model

Splitting the Car_Make column into Car_Make and Car_Model provides a clearer and more structured dataset, enabling easier analysis and better feature engineering. Separating these components allows for more specific insights, such as analysing trends or prices based on the car make or model individually. It also enhances the predictive power of machine learning models by providing distinct features that capture unique patterns, while adhering to best practices for clean and organized data



~~~
Mutating the name column into car make and car model
```{r}
usedcars <-extract(usedcars,Car_Make,c("Car_Make","Car_Model"), "([^ ]+) (.*)")
head(usedcars,5)
~~~

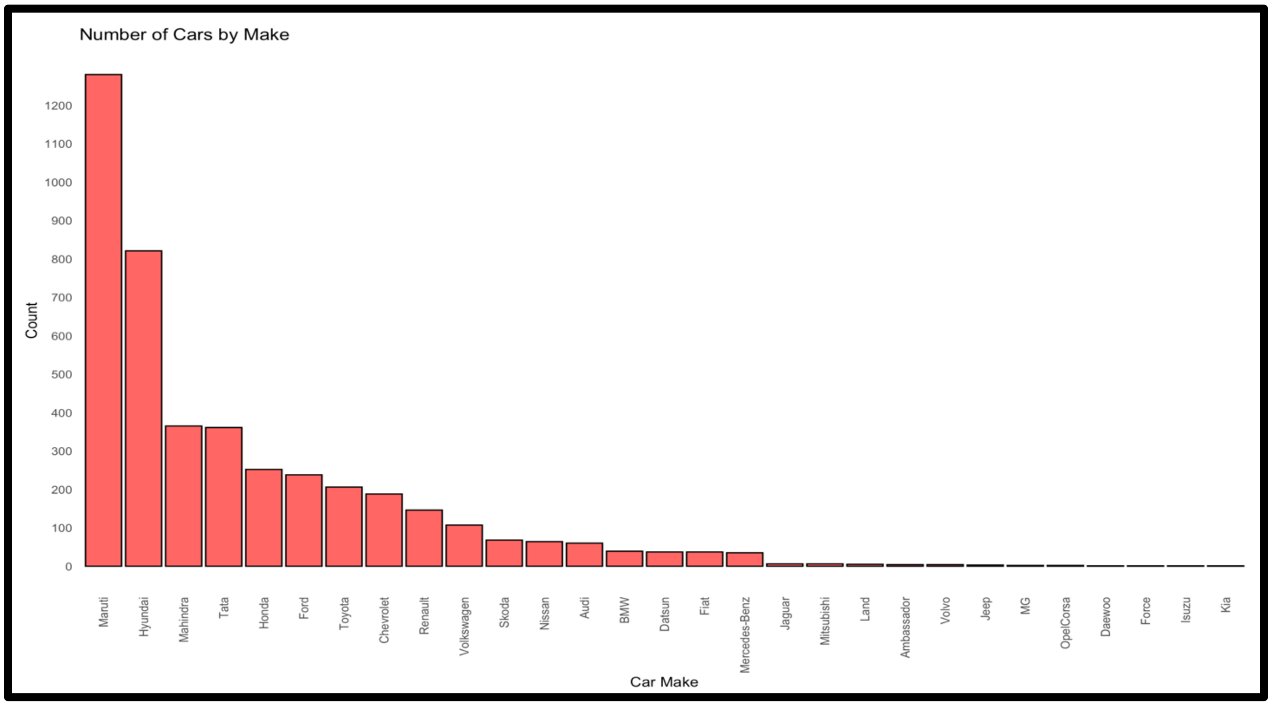
After splitting:



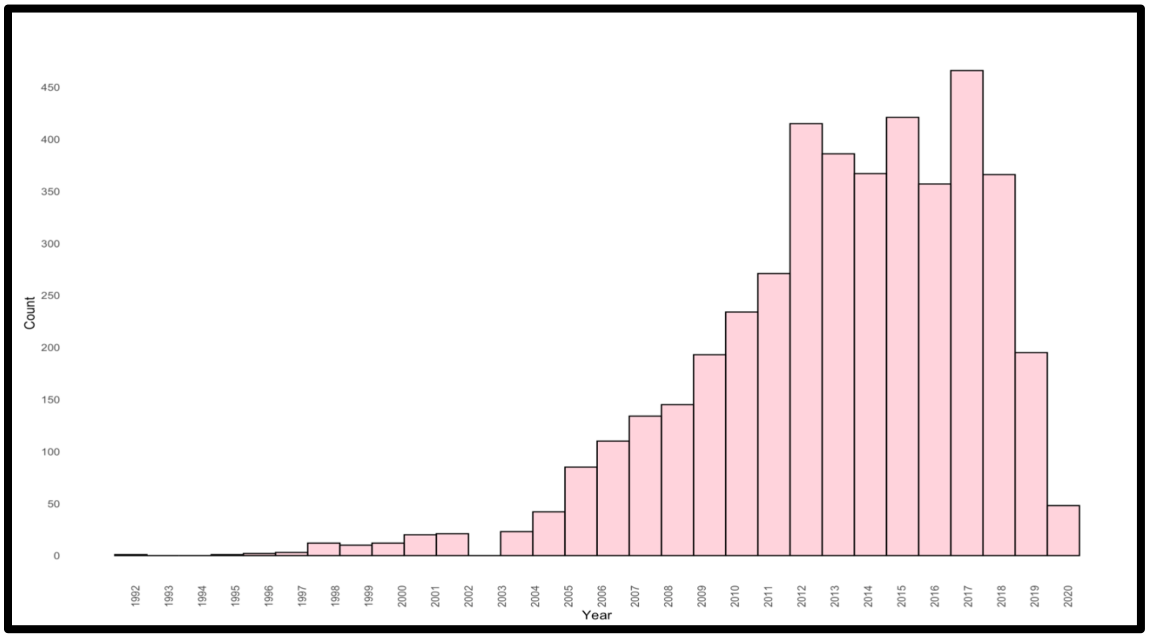
| | Car_Make | Car_Model | Year | Price | Mileage | Fuel_Type | Seller_Type | Transmission | Num_Owners |
|---|---|---|---|---|---|---|---|---|---|
| | <chr> | <chr> | <int> | <int> | <int> | <chr> | <chr> | <chr> | <chr> |
| 1 | Maruti | 800 AC | 2007 | 60000 | 70000 | Petrol | Individual | Manual | First Owner |
| 2 | Maruti | Wagon R LXI Minor | 2007 | 135000 | 50000 | Petrol | Individual | Manual | First Owner |
| 3 | Hyundai | Verna 1.6 SX | 2012 | 600000 | 100000 | Diesel | Individual | Manual | First Owner |
| 4 | Datsun | RediGO T Option | 2017 | 250000 | 46000 | Petrol | Individual | Manual | First Owner |
| 5 | Honda | Amaze VX i-DTEC | 2014 | 450000 | 141000 | Diesel | Individual | Manual | Second Owner |

## 2.3 Exploratory Data Analysis

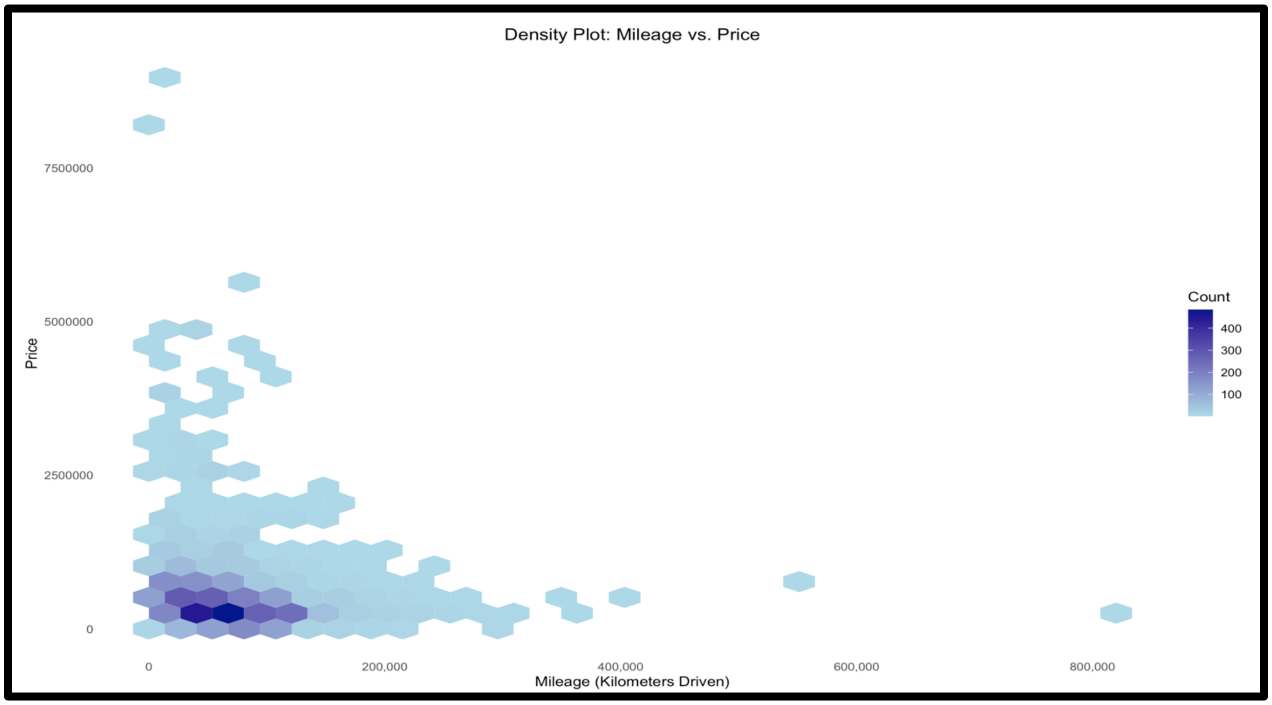
1. From fig-1, it can be visualized that Maruti is the most occurred Brand of dataset.



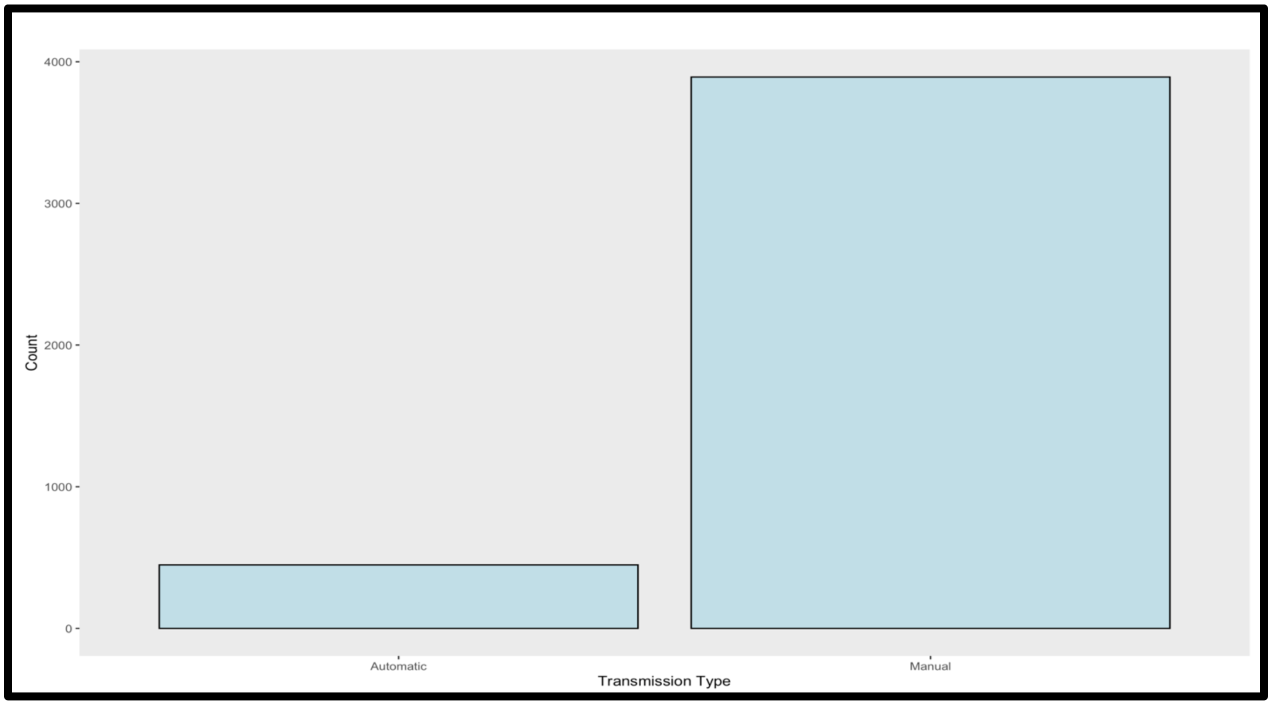
2. From fig-2, it can be visualized that years between 1992-1997 have very few cars in the dataset



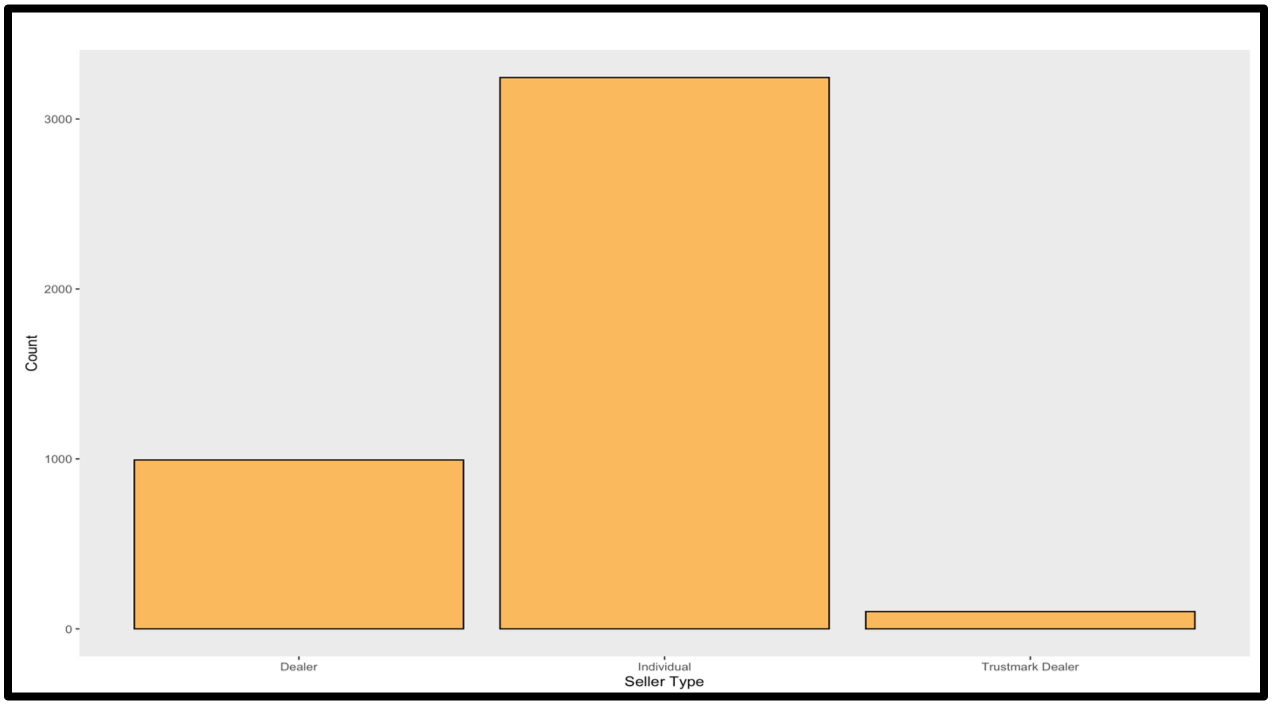
3. From fig-3 it can be visualized that most cars are with lower Mileage and Price in the dataset



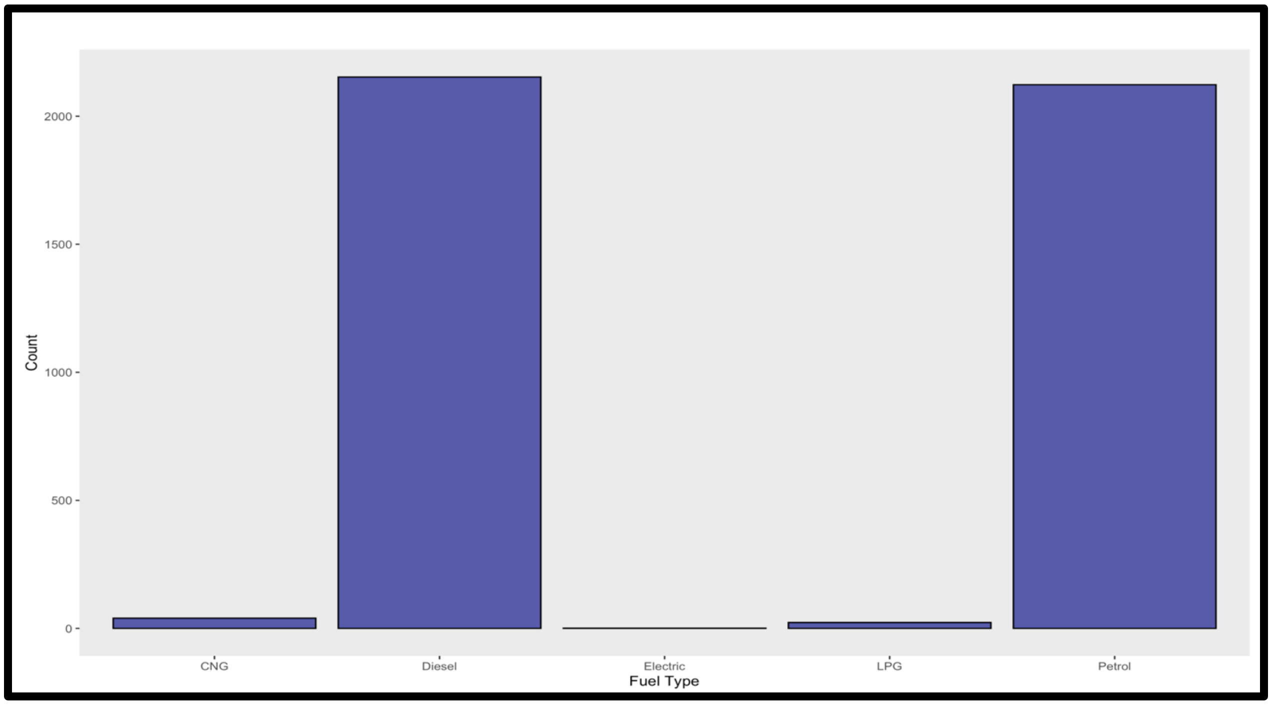
4. From fig-4 shows that most cars are manual in the dataset



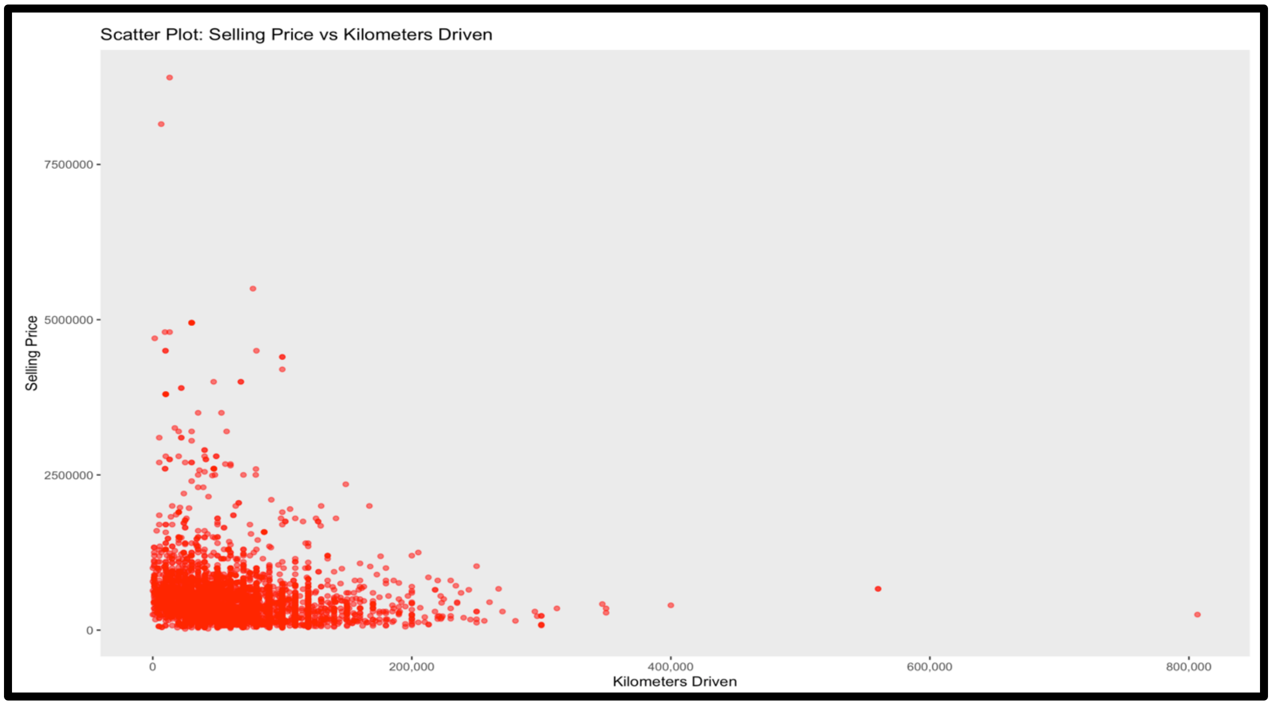
5. From fig-5 we can see that most cars in the dataset are Individuals



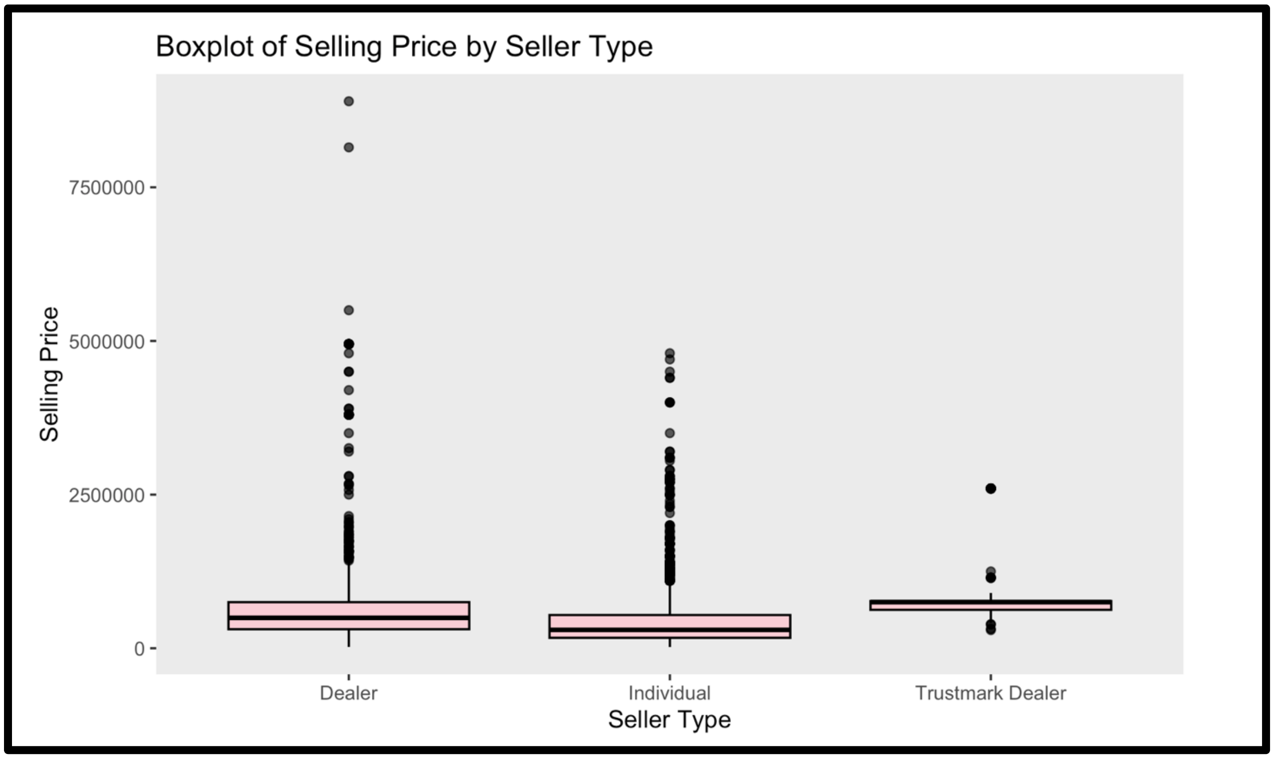
6. From fig-6 Diesel and Petrol is the most occurred fuel type in the dataset



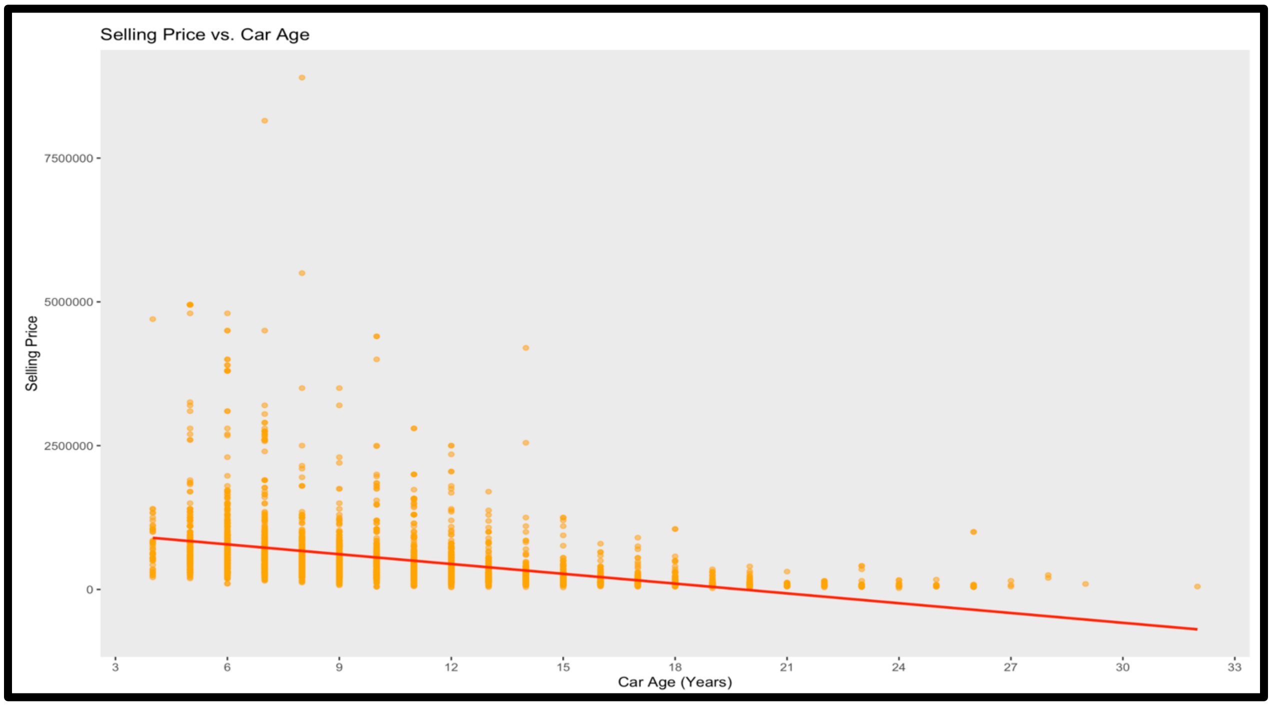
7. Fig-7 shows that most cars are low-priced and less-driven



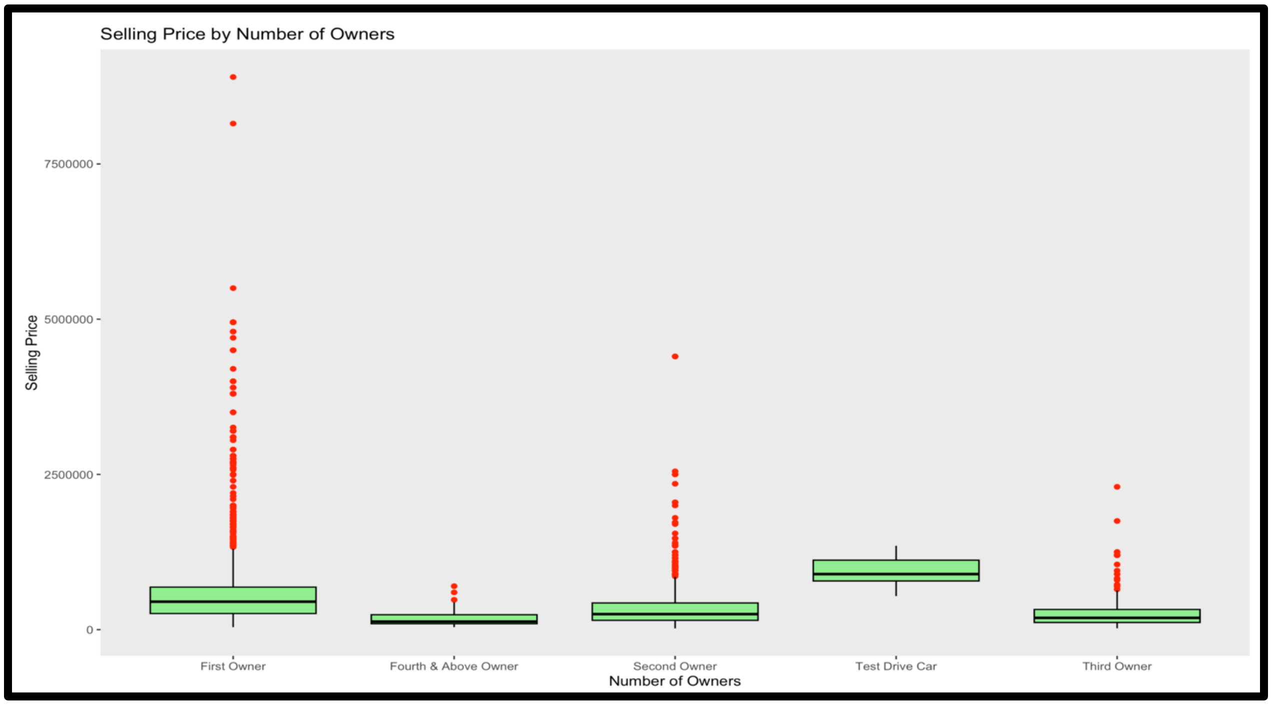
8. Fig-8 shows that cars sold by individuals are cheaper than other seller types



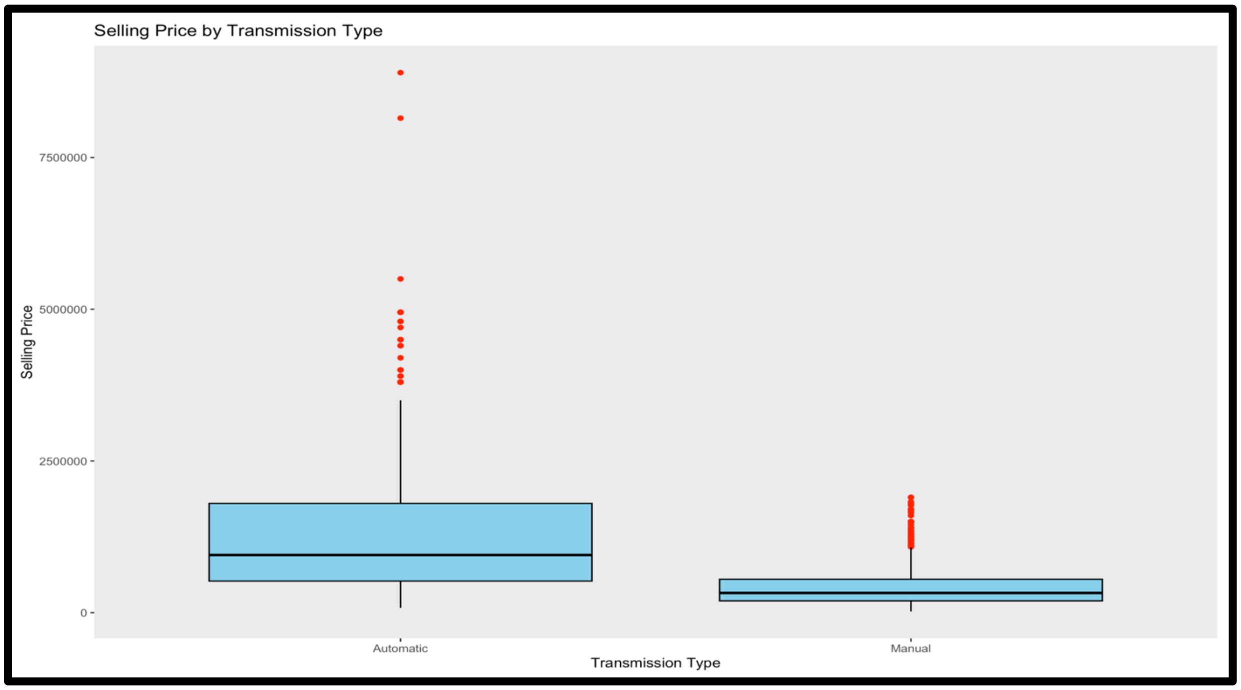
9. Fig-9 shows that newer cars are priced higher and vice-versa for older cars



10. Fig-10 shows that the price decreases with increasing number of ownerships

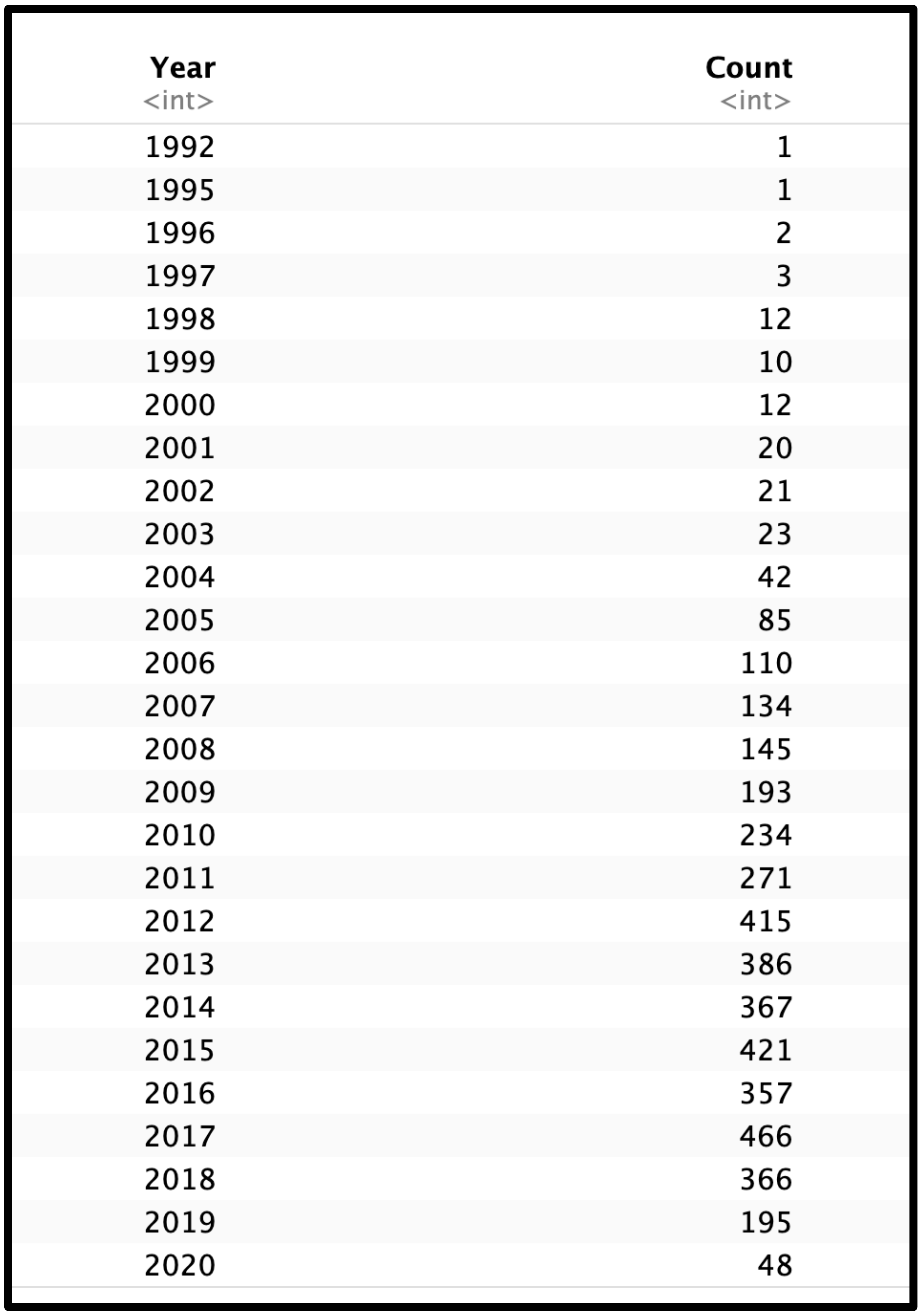


11.<u>From fig-11, it can be visualized that Automatic cars are more expensive than manual cars</u>



## 2.4 Handling the Outliers

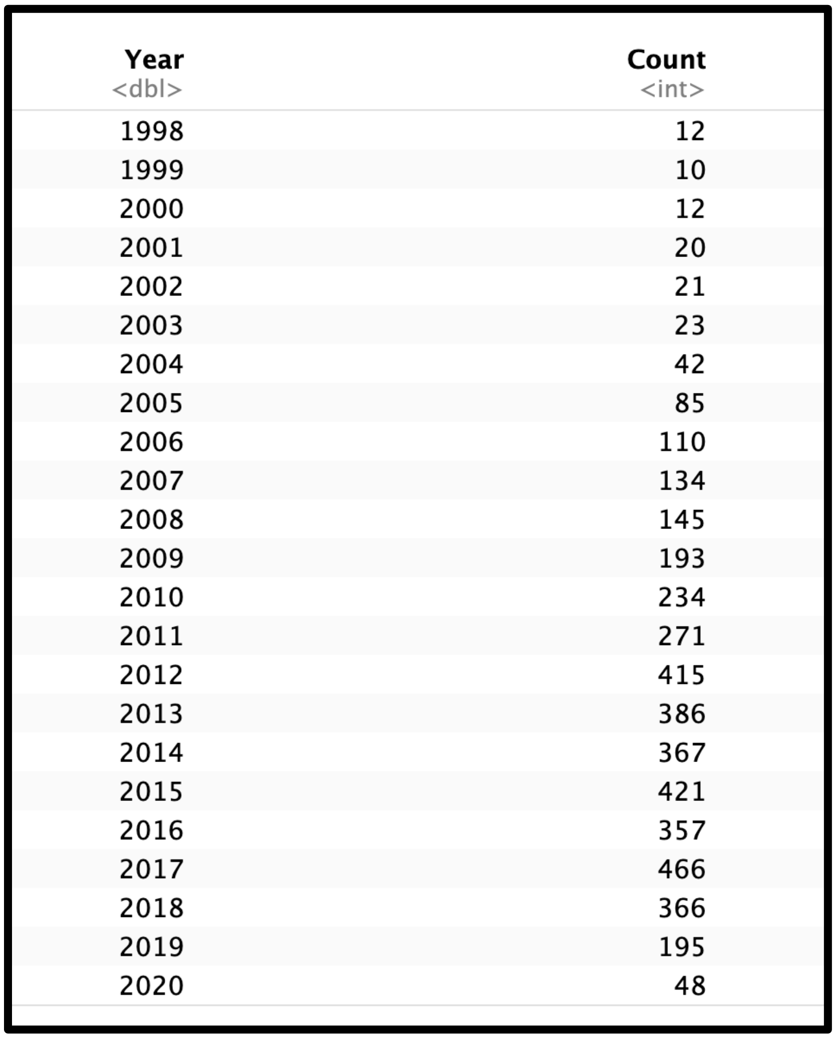
### 2.4.1 Removing years with just single-digit car units in it:



| Year | Count |
|---|---|
| <int> | <int> |
| 1992 | 1 |
| 1995 | 1 |
| 1996 | 2 |
| 1997 | 3 |
| 1998 | 12 |
| 1999 | 10 |
| 2000 | 12 |
| 2001 | 20 |
| 2002 | 21 |
| 2003 | 23 |
| 2004 | 42 |
| 2005 | 85 |
| 2006 | 110 |
| 2007 | 134 |
| 2008 | 145 |
| 2009 | 193 |
| 2010 | 234 |
| 2011 | 271 |
| 2012 | 415 |
| 2013 | 386 |
| 2014 | 367 |
| 2015 | 421 |
| 2016 | 357 |
| 2017 | 466 |
| 2018 | 366 |
| 2019 | 195 |
| 2020 | 48 |

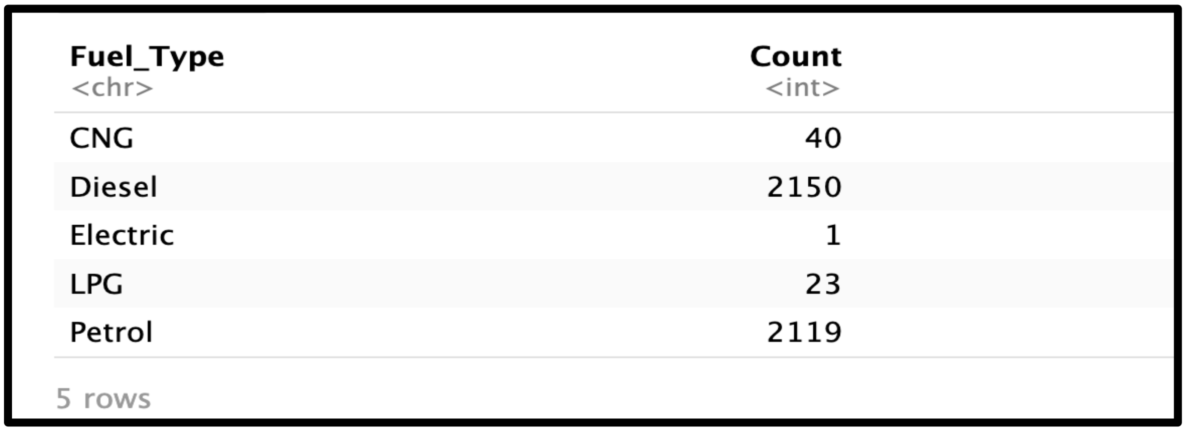
We found that the years 1992 to 2003 have very few cars, which could limit the model's ability to learn patterns for those years. To resolve this, the years starting from 1992 to 1997 were removed from the dataset, as they represent low-frequency data points. After filtering out these years, the count of cars sold per year is recalculated and displayed again, ensuring that the dataset has sufficient representation across years for training and testing the model. This helps ensure a balanced dataset and improves model performance by avoiding bias from low frequency categories.

**After removing years 1992-1997:**



| Year <dbl> | Count <int> |
|---|---|
| 1998 | 12 |
| 1999 | 10 |
| 2000 | 12 |
| 2001 | 20 |
| 2002 | 21 |
| 2003 | 23 |
| 2004 | 42 |
| 2005 | 85 |
| 2006 | 110 |
| 2007 | 134 |
| 2008 | 145 |
| 2009 | 193 |
| 2010 | 234 |
| 2011 | 271 |
| 2012 | 415 |
| 2013 | 386 |
| 2014 | 367 |
| 2015 | 421 |
| 2016 | 357 |
| 2017 | 466 |
| 2018 | 366 |
| 2019 | 195 |
| 2020 | 48 |

### 2.4.2 Removal of "Electric" in fuel_type:

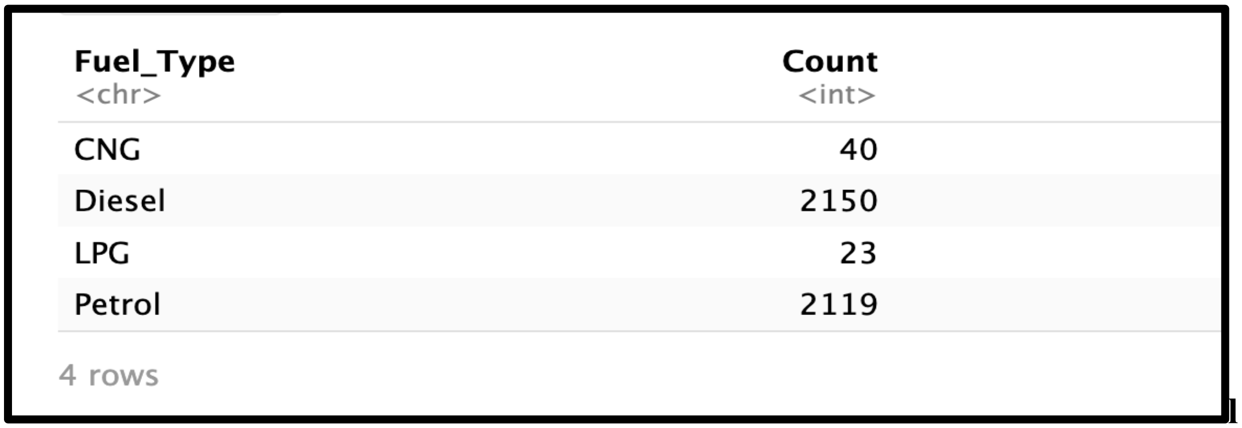


| Fuel_Type <chr> | Count <int> |
|---|---|
| CNG | 40 |
| Diesel | 2150 |
| Electric | 1 |
| LPG | 23 |
| Petrol | 2119 |

5 rows

Upon grouping cars by the fuel type, we found there was only 1 representation for the fuel type – Electric. Since this is an insignificant representation in the dataset, it was removed. The updated dataset is saved as ***data_cleaned*** and reassigned to usedcars. Finally, the count of cars for each remaining fuel type is recalculated and displayed, ensuring that the dataset only includes fuel types with sufficient data for reliable modelling. This step helps maintain data consistency and ensures better model performance.

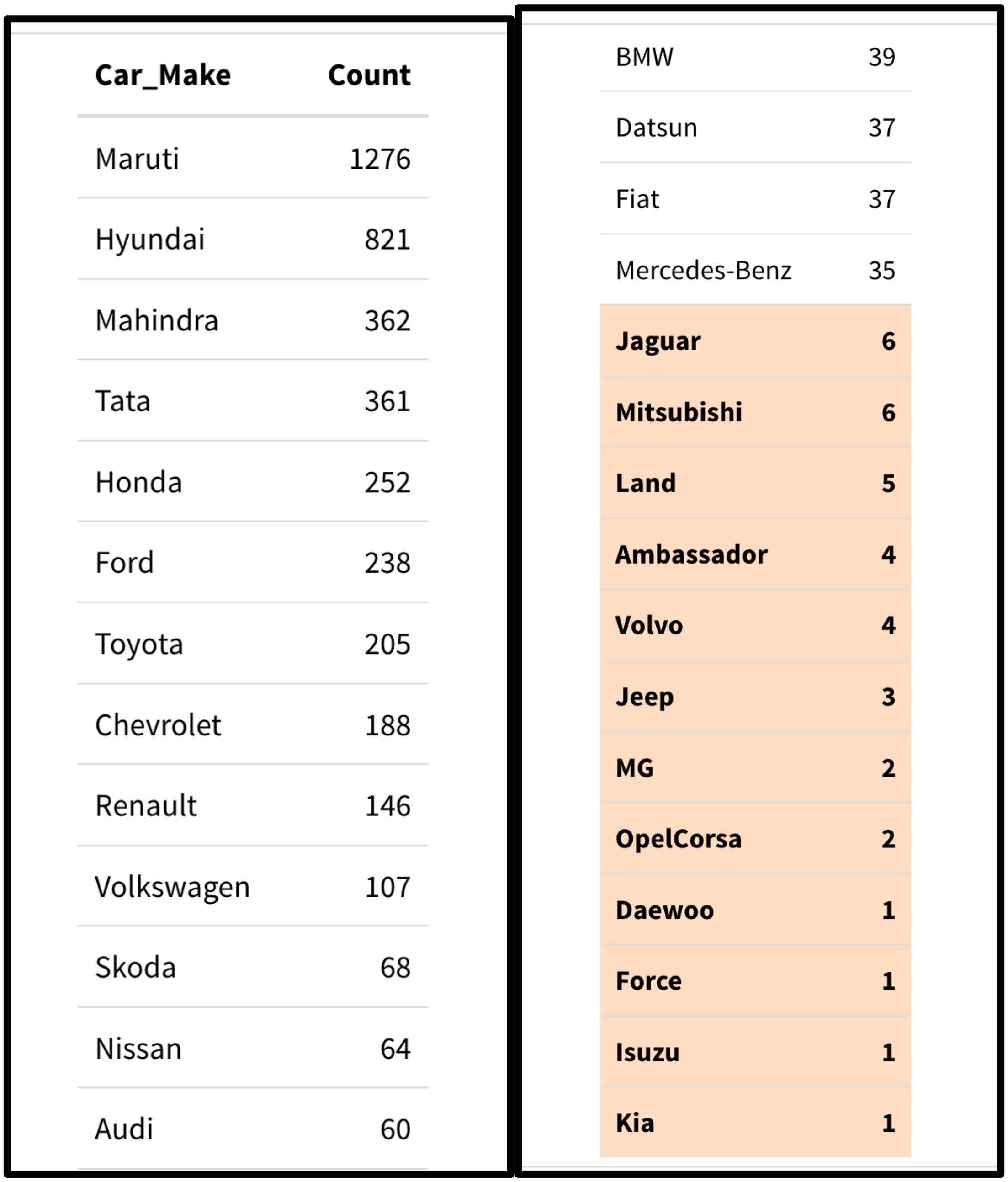
**After removal:**



| Fuel_Type <chr> | Count <int> |
|---|---|
| CNG | 40 |
| Diesel | 2150 |
| LPG | 23 |
| Petrol | 2119 |

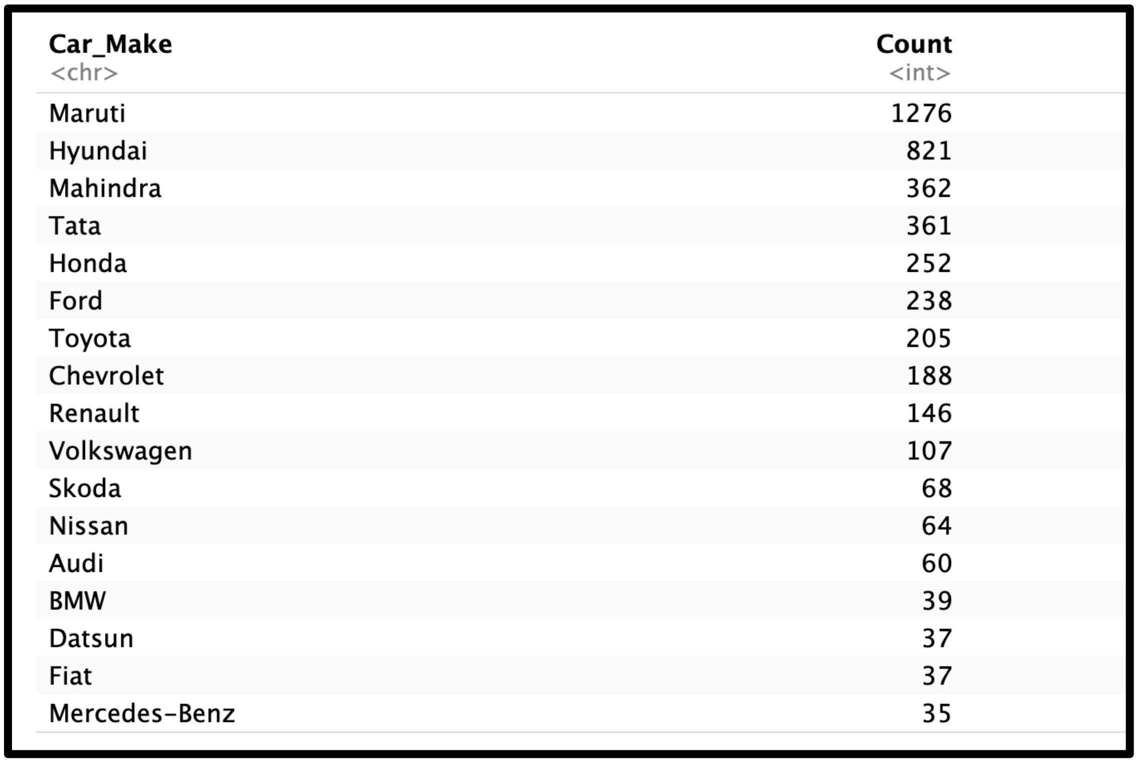
4 rows

### 2.4.3 Removal of car brands with cars less than 10



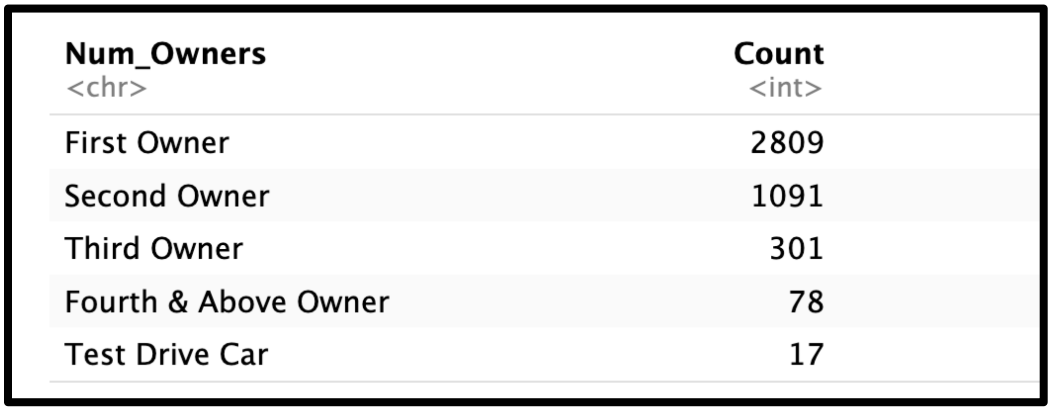
| Car_Make | Count |
|---|---|
| Maruti | 1276 |
| Hyundai | 821 |
| Mahindra | 362 |
| Tata | 361 |
| Honda | 252 |
| Ford | 238 |
| Toyota | 205 |
| Chevrolet | 188 |
| Renault | 146 |
| Volkswagen | 107 |
| Skoda | 68 |
| Nissan | 64 |
| Audi | 60 |
| BMW | 39 |
| Datsun | 37 |
| Fiat | 37 |
| Mercedes-Benz | 35 |
| **Jaguar** | **6** |
| **Mitsubishi** | **6** |
| **Land** | **5** |
| **Ambassador** | **4** |
| **Volvo** | **4** |
| **Jeep** | **3** |
| **MG** | **2** |
| **OpelCorsa** | **2** |
| **Daewoo** | **1** |
| **Force** | **1** |
| **Isuzu** | **1** |
| **Kia** | **1** |

To analyze the representation of different car makes in the dataset, the number of cars for each Car_Make was calculated and organized in descending order of frequency. Brands with lower than 10 entries were identified and these low-frequency car makes were removed for further consideration, as they could impact the model's ability to generalize. This step ensures better data consistency and helps decide whether to group or remove underrepresented categories during preprocessing.



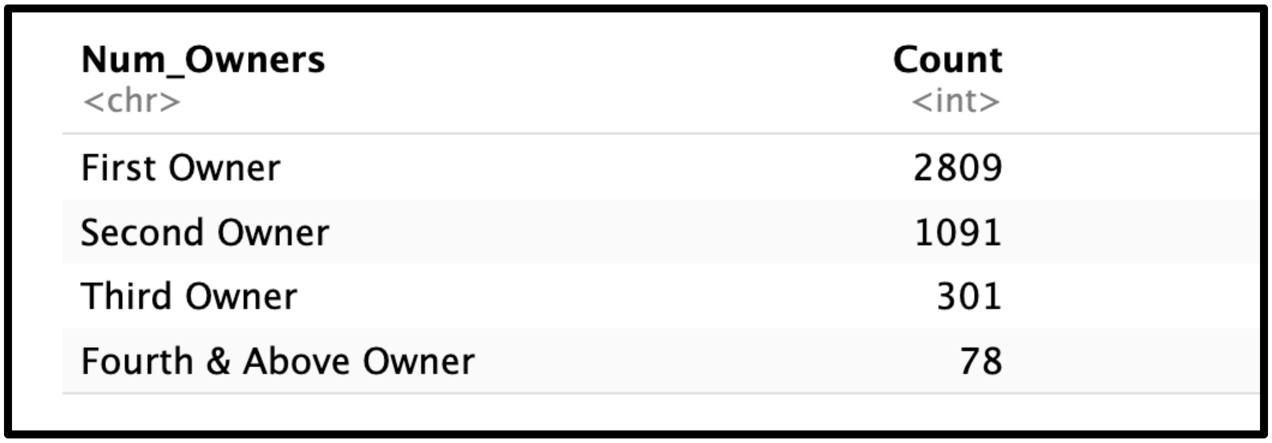
| Car_Make <chr> | Count <int> |
|---|---|
| Maruti | 1276 |
| Hyundai | 821 |
| Mahindra | 362 |
| Tata | 361 |
| Honda | 252 |
| Ford | 238 |
| Toyota | 205 |
| Chevrolet | 188 |
| Renault | 146 |
| Volkswagen | 107 |
| Skoda | 68 |
| Nissan | 64 |
| Audi | 60 |
| BMW | 39 |
| Datsun | 37 |
| Fiat | 37 |
| Mercedes-Benz | 35 |

### 2.4.4 Remove Test Drive Car from the Num of Owners column:



| Num_Owners <chr> | Count <int> |
|---|---|
| First Owner | 2809 |
| Second Owner | 1091 |
| Third Owner | 301 |
| Fourth & Above Owner | 78 |
| Test Drive Car | 17 |

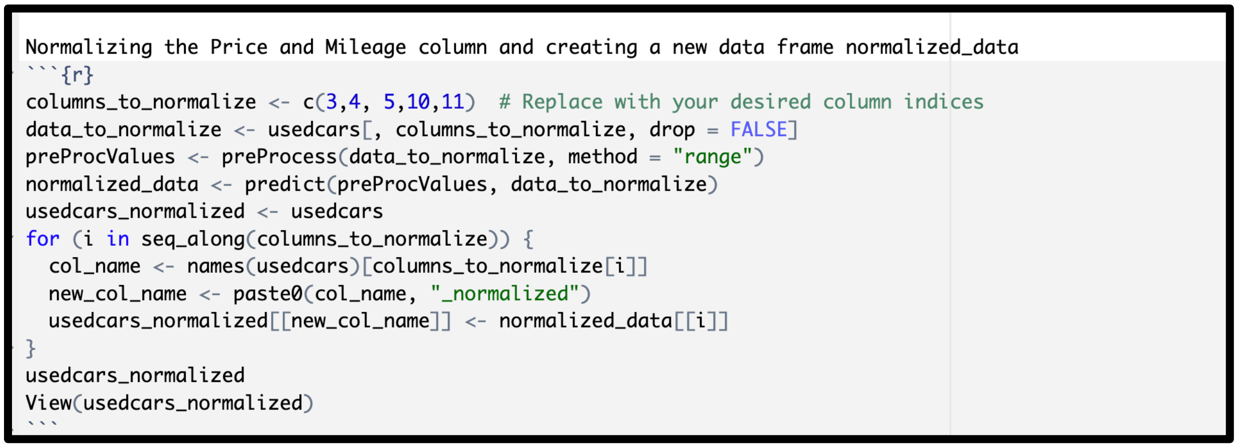
We remove the cars belonging to the ***Test Drive Car*** category in the ***Num of Columns*** column as the representation here is insufficient with the number of cars falling under this category is just 17. Including such data can introduce noise, skew the analysis, and reduce the reliability of the model's predictions.



| Num_Owners | Count |
|---|---|
| <chr> | <int> |
| First Owner | 2809 |
| Second Owner | 1091 |
| Third Owner | 301 |
| Fourth & Above Owner | 78 |

## 2.5 Data pre-processing

Pre-processing is a Data Mining technique that involves converting raw data into a comprehensible format. There is often a lack of specific activity or trend data, and many inaccurate facts are included in real-world data. Consequently, this may result in poor-quality data collection, and, in turn, poor-quality models constructed from the data. Such problems can be resolved by pre-processing the data.

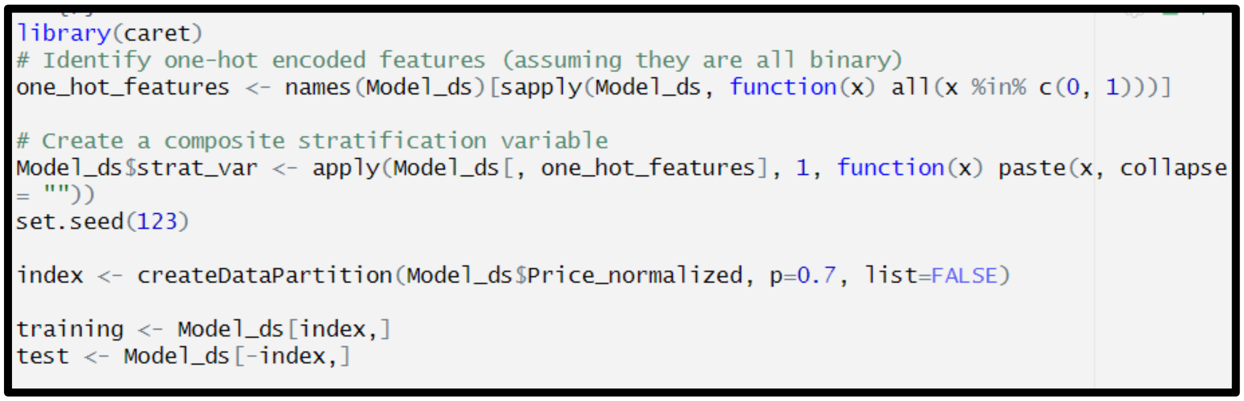

~~~
Normalizing the Price and Mileage column and creating a new data frame normalized_data
```{r}
columns_to_normalize <- c(3,4, 5,10,11)  # Replace with your desired column indices
data_to_normalize <- usedcars[, columns_to_normalize, drop = FALSE]
preProcValues <- preProcess(data_to_normalize, method = "range")
normalized_data <- predict(preProcValues, data_to_normalize)
usedcars_normalized <- usedcars
for (i in seq_along(columns_to_normalize)) {
  col_name <- names(usedcars)[columns_to_normalize[i]]
  new_col_name <- paste0(col_name, "_normalized")
  usedcars_normalized[[new_col_name]] <- normalized_data[[i]]
}
usedcars_normalized
View(usedcars_normalized)
```
~~~

We normalized these specific columns: ***mileage, price, Year, Price per KM, Car Age*** from the ***usedcars*** dataset to ensure that their values fall within a consistent range, typically between 0 and 1. This is done because features with vastly different scales can disproportionately influence machine learning models, leading to biased predictions. We first specify the columns to normalize using their indices and extract these columns into ***data_to_normalize***. The ***preProcessfunction*** from the caret library is used with the "range" method to compute the necessary scaling and transformation parameters (***preProcValues***). Using the predict function, we apply these transformations, generating a new set of normalized values. Finally, we add these normalized columns back to the original dataset (***usedcars_normalized***) with descriptive column names (e.g., ***mileage_normalized, price_normalized***). This ensures that both the original and normalized data are retained for analysis or modeling. We then create a new data set by removing the car model, Year, Price, Age, Price per KM and Mileage column named ***Model_ds.*** This data set is aimed to just have normalized columns for implementing One hot encoding in the later steps.

## 2.6 One-hot encoding

we transformed the dataset (***Model_ds***) to convert categorical variables into numerical dummy variables, making it suitable for machine learning algorithms. This is necessary because most machine learning models cannot directly process categorical data as they require numerical input. Using the ***dummyVars*** function from the caret library with the formula "~.", we specified that all columns in the dataset should be included in the transformation. The f***ullRank = TRUE*** option ensures that redundant dummy variables are excluded, preventing multicollinearity. The predict function then applies this transformation, creating a new dataset (***trsf***) with numerical representations of the original categorical variables. Finally, the transformed dataset replaces the original (***Model_ds***).

## 2.7 Data Partitioning

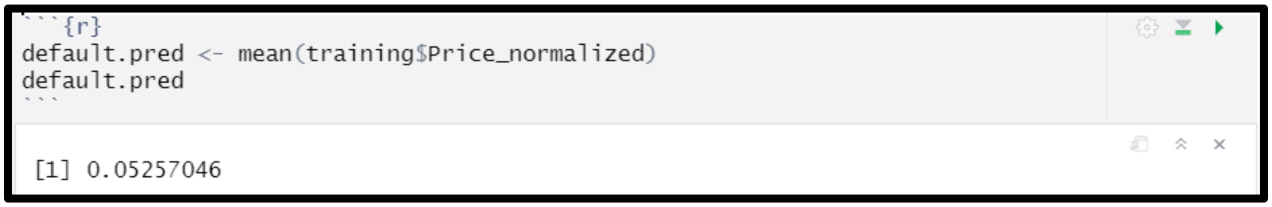


```
library(caret)
# Identify one-hot encoded features (assuming they are all binary)
one_hot_features <- names(Model_ds)[sapply(Model_ds, function(x) all(x %in% c(0, 1)))]

# Create a composite stratification variable
Model_ds$strat_var <- apply(Model_ds[, one_hot_features], 1, function(x) paste(x, collapse
= ""))
set.seed(123)

index <- createDataPartition(Model_ds$Price_normalized, p=0.7, list=FALSE)

training <- Model_ds[index,]
test <- Model_ds[-index,]
```

The code is designed to split the dataset into training and testing subsets while maintaining the distribution of the target variable, ***Price_normalized***, across both subsets. The process begins by identifying binary (one-hot encoded) features in the dataset and creating a composite stratification variable (***strat_var***) that combines these features into a unique string for each row. It is done to ensure that the training and testing dataset maintains a similar distribution of the target variable and considering the combination of binary features.

This ensures that the stratification accounts for the combinations of these binary features. Using the Caret package's ***createDataPartition*** function, the data is then split into a training set (70%) and a testing set (30%), with stratification based on the ***Price_normalized*** variable. Stratification ensures that the proportion of different values in ***Price_normalized*** is preserved in both subsets, reducing potential bias, and improving model generalizability. The dimensions of the resulting subsets are verified, and summaries of ***Price_normalized*** are generated to confirm consistent distributions between training and testing sets. This approach is crucial in machine learning to create balanced datasets for training and evaluation, particularly when dealing with imbalanced or categorical target variables.

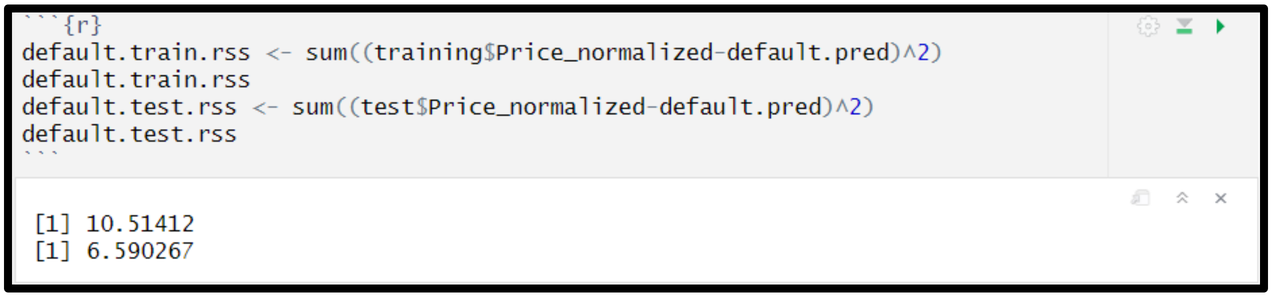
To calculate the default prediction for the dependent variable:



~~~
```{r}
default.pred <- mean(training$Price_normalized)
default.pred
```
~~~

```
[1] 0.05257046
```

The ***default.pred*** calculates the mean of the normalized price from the training dataset which is approximately 0.053. This mean value serves as the most basic prediction model - essentially predicting the same average price for all cases. This naive approach assumes no relationship between features and target variable, predicting the same value (mean) for all instances. When we calculate the Residual Sum of Squares (RSS) for both training and test datasets using this mean prediction, we get a measure of how much our predictions deviate from the actual values.



```{r}
default.train.rss <- sum((training$Price_normalized-default.pred)^2)
default.train.rss
default.test.rss <- sum((test$Price_normalized-default.pred)^2)
default.test.rss
```

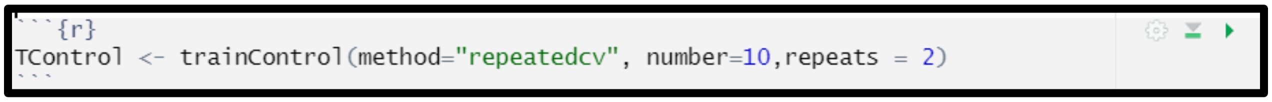
```
[1] 10.51412
[1] 6.590267
```

Using this baseline, the code then computes the residual sum of squares (RSS) for both the training and test datasets. It measures how far the actual values in the training set and test set deviate from the baseline prediction. The Residual Sum of Squares (RSS) is then computed for both training (***default.train.rss)*** and test (***default.test.rss***) datasets to measure prediction error. These baseline RSS values serve as benchmarks to evaluate more advanced models. Comparing RSS across training and test sets highlights how well models improve over this naive approach and helps assess generalization performance.

By comparing ***train.rss*** and ***test.rss*** of a trained model with these baseline values, we can assess whether our model improves upon this naive approach.

## 2.7 Cross-validation and repeats

We set out to configure the settings of the training control of the model using the Caret '***trainControl'*** function to set up cross validation for model training.



```{r}
TControl <- trainControl(method="repeatedcv", number=10,repeats = 2)
```

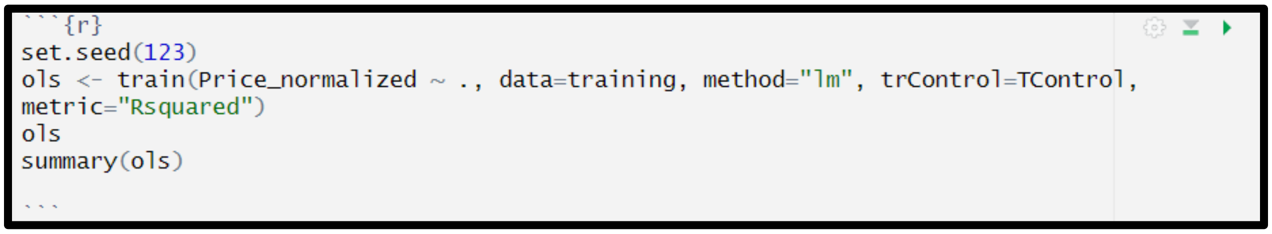
The code sets up a resampling strategy using repeated Cross-Validation (Repeated CV) with 10 folds and 2 repetitions. In this approach, the dataset is split into 10 equal parts (folds), and the model is trained and tested iteratively on different combinations of these folds. The entire process is repeated twice with different random splits to ensure robustness. Repeated CV improves upon standard cross-validation by reducing variability in performance metrics caused by random splits, providing more reliable estimates of model generalization ability. Using repeats of 2 balances computational efficiency with enhanced reliability, making it particularly useful for evaluating models in scenarios where stability and generalization are critical.

During hyperparameter tuning, metrics such as RMSE and $R^2$ may diverge across different configurations due to randomness in data splits during standard cross-validation. Repeating the cross-validation process twice with different random partitions reduces variability in these metrics, providing more robust and reliable estimates of model performance. This ensures that the selected hyperparameters generalize well to unseen data while balancing computational efficiency with improved stability in performance evaluation.

# 3. MODELS

## 3.1 Linear Regression

We performed a linear regression model (***lm***) to predict ***Price_normalized*** using all available features in the training dataset. The model was implemented using the ***caret*** package in R, with a specified random seed (***set.seed(123)***) to ensure reproducibility. We used a cross-validation control object (***TControl***) to evaluate the model's performance, optimizing for the ***Rsquared*** metric to assess how well the model explains the variance in the target variable. After fitting the model, we printed the results with ols and generated a detailed statistical summary using ***summary***(***ols***) to analyze the coefficients and overall model performance.



~~~
```{r}
set.seed(123)
ols <- train(Price_normalized ~ ., data=training, method="lm", trControl=TControl,
metric="Rsquared")
ols
summary(ols)

```
~~~

The results of the linear regression model indicate its performance in predicting ***Price_normalized.*** The model achieved a **Root Mean Squared Error (RMSE)** of 0.2732, reflecting the average magnitude of prediction errors. The **R-squared** value is 0.6001, meaning the model explains approximately 60% of the variance in the target variable. Additionally, the **Mean Absolute Error (MAE)** is 0.0198, representing the average absolute difference between predicted and actual values, and the **Mean Squared Error (MSE)** is 0.0011, indicating the squared average of the errors.

| **Term** | **Coefficient** |
|---|---|
| (Intercept) | 0.123594876 *** |
| Car_MakeBMW | 0.093976001 *** |
| Car_MakeChevrolet | -0.127540406 *** |
| Car_MakeDatsun | -0.136003965 *** |
| Car_MakeFiat | -0.120709311 *** |
| Car_MakeFord | -0.105575554 *** |
| Car_MakeHonda | -0.102006288 *** |
| Car_MakeHyundai | -0.110784399 *** |
| Car_MakeMahindra | -0.10110171 *** |
| Car_MakeMaruti | -0.114529058 *** |
| Car_MakeMercedes.Benz | 0.099517052 *** |
| Car_MakeNissan | -0.113220357 *** |
| Car_MakeRenault | -0.122574113 *** |
| Car_MakeSkoda | -0.11323672 *** |
| Car_MakeTata | -0.123542568 *** |
| Car_MakeToyota | -0.063627792 *** |
| Car_MakeVolkswagen | -0.115239072 *** |
| Fuel_TypeDiesel | 0.021427935 ** |
| Fuel_TypeLPG | 0.004821983 |
| Fuel_TypePetrol | 0.00018375 |
| Seller_TypeIndividual | -0.001394387 |
| Seller_TypeTrustmark.Dealer | 0.031644884 *** |
| TransmissionManual | -0.036867514 *** |
| Num_OwnersFourth...Above.Owner | 0.002441951 |
| Num_OwnersSecond.Owner | -0.004467938 ** |
| Num_OwnersThird.Owner | -0.002808451 |
| Year_normalized | 0.097029217 *** |
| Mileage_normalized | -0.094205791 *** |
| **Car_Age_normalized** | NA |
| Price_per_KM_normalized | -0.03715646 |

The variable ***Car_Age_normalized*** was created as:

***Car_Age_normalized* = *Current_Year* − *Year_normalized***

This means that C***ar_Age_normalized*** is perfectly correlated with ***Year_normalized* ,** as they are mathematically linked.

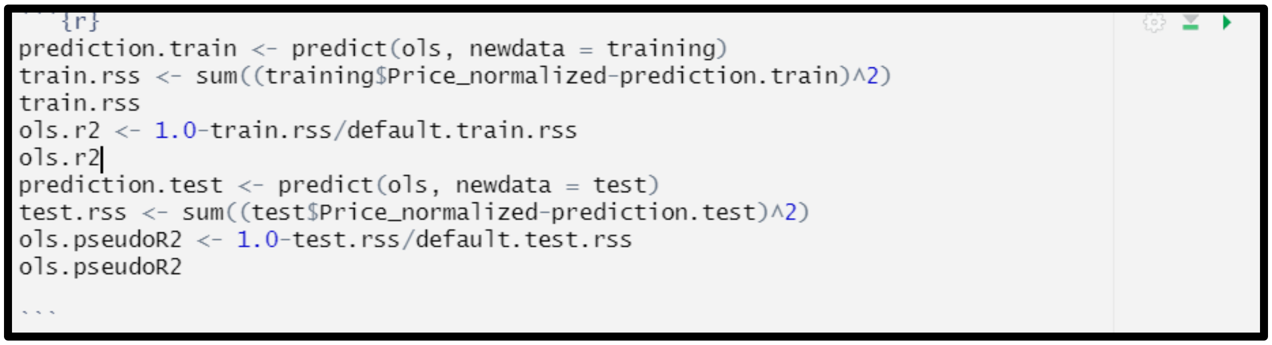
The coefficient for ***Car_Age_normalized*** is assigned an NA in the linear regression model because of perfect multicollinearity with another variable, specifically

***Year_normalized***. This issue arises because ***Car_Age_normalized*** is a feature-engineered variable derived from subtracting the ***Year_normalized*** (representing the normalized manufacturing year of the car) from the current year.

As a result, ***Car_Age_normalized*** and ***Year_normalized*** are linearly dependent and convey overlapping information about the car's age. When multicollinearity exists, the design matrix becomes singular or near-singular, making it impossible to compute unique regression coefficients for all variables.

To handle this, R automatically assigns an NA to one of the redundant variables (in this case, ***Car_Age_normalized***) to prevent instability in the regression solution.

Linear regression models are extremely sensitive to multicollinearity, where independent variables are highly correlated with one another. This sensitivity arises because multicollinearity makes it difficult for the model to isolate the unique contribution of each predictor, leading to unstable and unreliable coefficient estimates.



~~~
```{r}
prediction.train <- predict(ols, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
train.rss
ols.r2 <- 1.0-train.rss/default.train.rss
ols.r2
prediction.test <- predict(ols, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
ols.pseudoR2 <- 1.0-test.rss/default.test.rss
ols.pseudoR2

```
~~~

The linear regression model's performance was evaluated using R$^2$ and pseudo- R$^2$ metrics, which measure how well the model explains variance in both training and test datasets compared to a baseline model predicting only the mean. For training data, $R^2$ =0.6733, indicating that approximately 67.33% of variance in Price_normalized is explained by the model. On test data, pseudo- $R^2$ =0.6723, showing similar performance on unseen data and suggesting good generalization ability. These metrics confirm that the linear regression model effectively captures patterns in the data without overfitting, making it suitable for predicting normalized car prices.

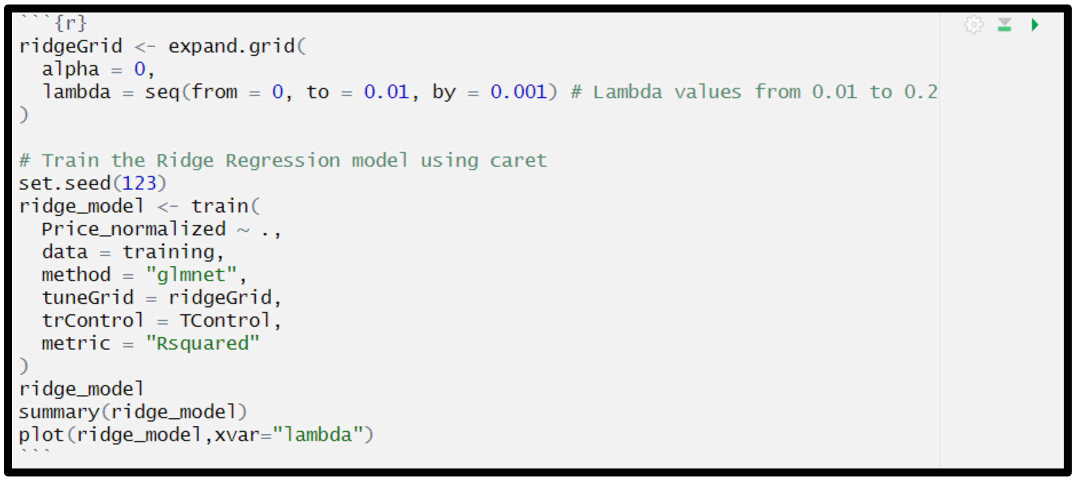
## 3.2 Ridge Regression

To address the issue of multicollinearity, Ridge Regression was employed. Ridge Regression resolves this by introducing a regularization term (lambda) into the cost function, which penalizes large coefficients and shrinks them toward zero. This reduces the impact of multicollinearity and stabilizes the model's predictions.

In this implementation:

The regularization term (lambda) was tuned over a sequence starting from 0.0 to 0.05 with increments of 0.001.

By adding a penalty proportional to the sum of squared coefficients, Ridge Regression minimizes overfitting while retaining all features in the model. This approach ensures that the model performs well even when predictors are highly correlated, as seen in this case where ***Car_Age_normalized*** and ***Year_normalized*** exhibited perfect multicollinearity. The Ridge model effectively stabilized coefficient estimates and improved generalization to unseen data.



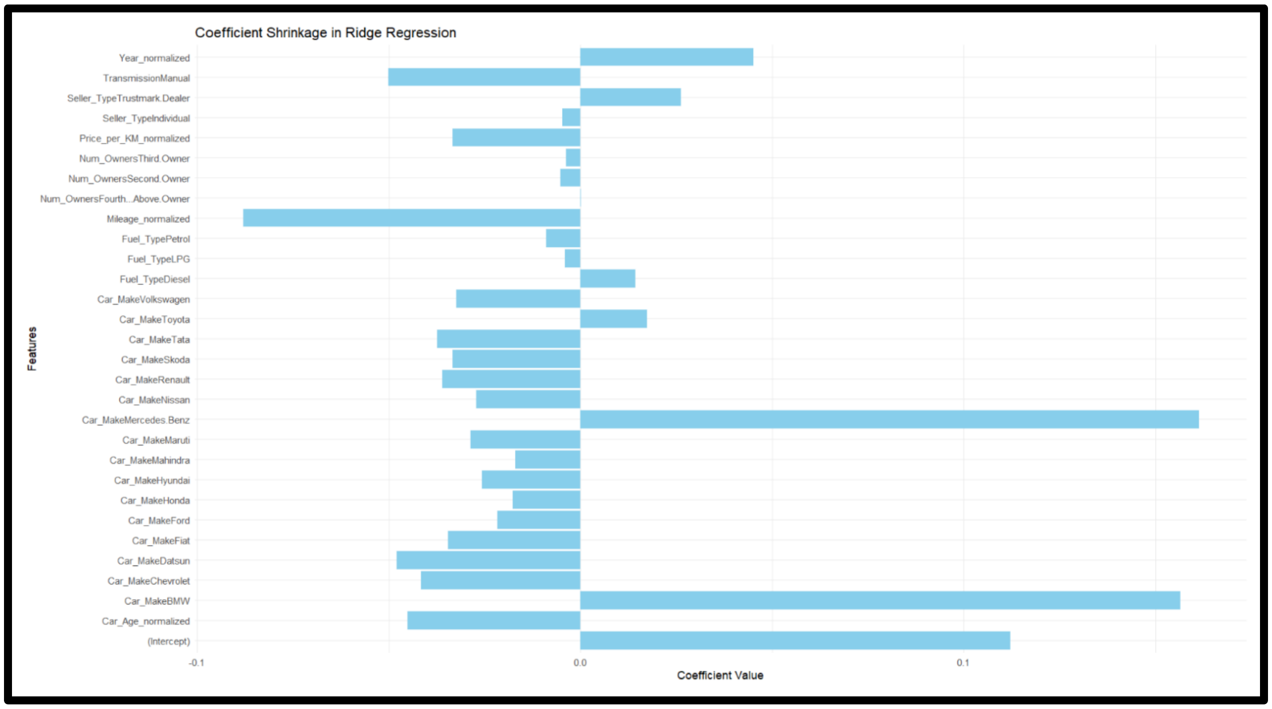
```{r}
ridgeGrid <- expand.grid(
  alpha = 0,
  lambda = seq(from = 0, to = 0.01, by = 0.001) # Lambda values from 0.01 to 0.2
)

# Train the Ridge Regression model using caret
set.seed(123)
ridge_model <- train(
  Price_normalized ~ .,
  data = training,
  method = "glmnet",
  tuneGrid = ridgeGrid,
  trControl = TControl,
  metric = "Rsquared"
)
ridge_model
summary(ridge_model)
plot(ridge_model,xvar="lambda")
```

The table below shows the tuning process for the hyperparameters to find the best set-up. The resulting Lambda is equal to 0.02.

| alpha | lambda | RMSE | Rsquared | MAE |
|---|---|---|---|---|
| 0 | 0 | 0.2694019 | 0.580466002 | 0.034156729 |
| 0 | 0.001 | 0.2694019 | 0.580466002 | 0.034156729 |
| **0** | **0.002** | **0.2694019** | **0.580466002** | **0.034156729** |
| 0 | 0.003 | 0.269403256 | 0.580454027 | 0.034157037 |
| 0 | 0.004 | 0.267673769 | 0.57846346 | 0.034084389 |
| 0 | 0.005 | 0.265835658 | 0.576952328 | 0.033987423 |
| 0 | 0.006 | 0.263996911 | 0.575842426 | 0.033880761 |
| 0 | 0.007 | 0.262169481 | 0.5749865 | 0.03376876 |
| 0 | 0.008 | 0.260363394 | 0.574297398 | 0.03365544 |
| 0 | 0.009 | 0.258579136 | 0.573726092 | 0.033542199 |
| 0 | 0.01 | 0.256819754 | 0.57323463 | 0.033430267 |

**Coefficient shrinkage in Ridge Regression:**

The provided plot illustrates the **coefficient shrinkage in Ridge Regression**, where the magnitudes of feature coefficients are reduced due to the regularization effect of the ridge penalty (L2 regularization). In ridge regression, the penalty term is added to the loss function to prevent overfitting by discouraging large coefficients.



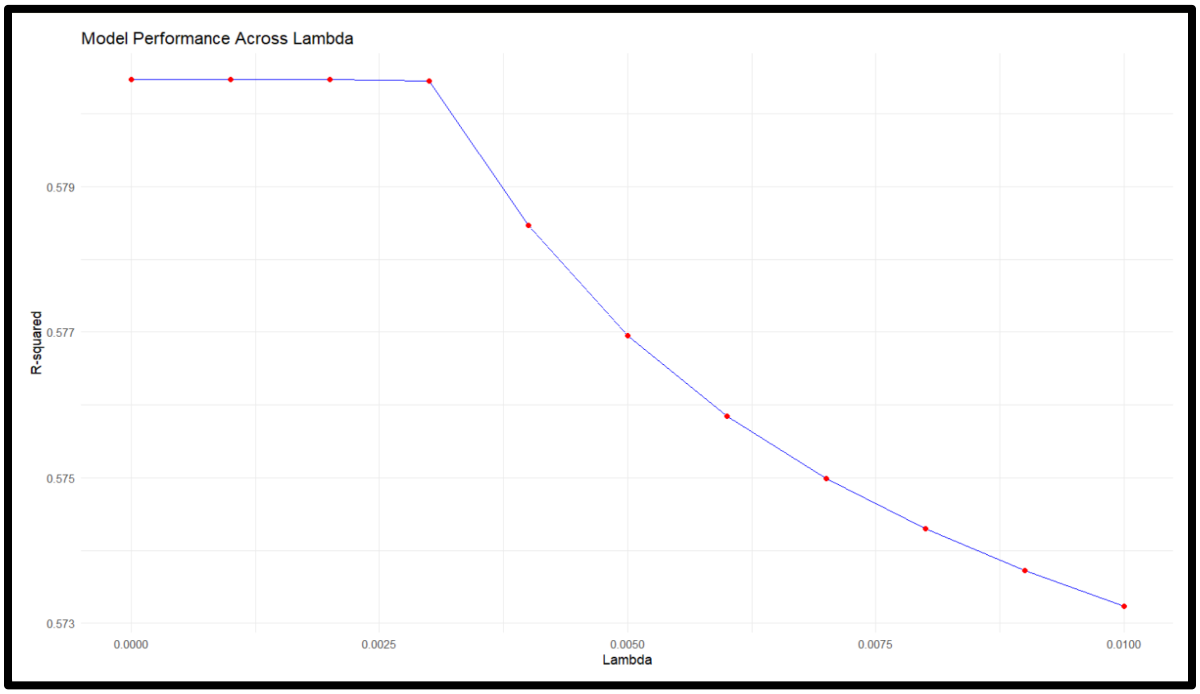
From the graph, we observe that:

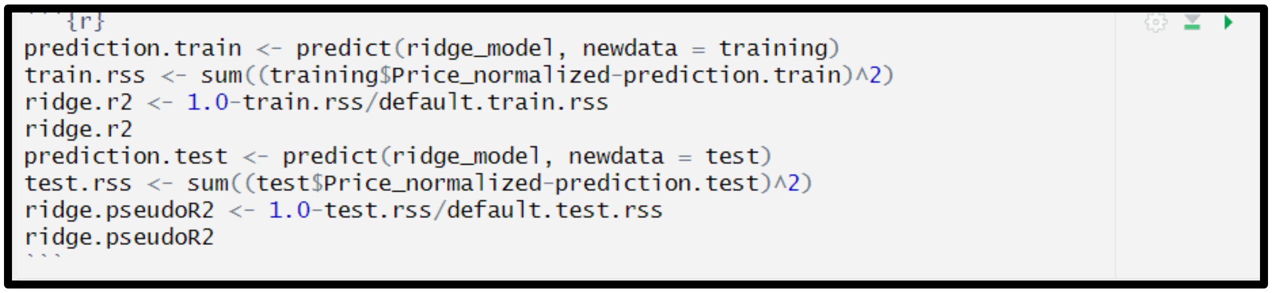
Most coefficients are close to zero, indicating that ridge regression has effectively shrunk their values. This suggests these features have a limited contribution to the model's predictive power.

Some features retain relatively larger coefficients, such as ***Car_MakeMercedes,Benz, Car_MakeBMW,*** and ***Car_Age_normalized***. These features likely have a stronger relationship with the target variable (***Price_normalized***) and are less penalized by the regularization.

Features like ***TransmissionManual*** and ***Mileage_normalized*** also have noticeable contributions, while others, such as certain car makes and owner types, have coefficients closer to zero, implying minimal impact.

we are plotting the hyperparameter tuning for the better representation of the process



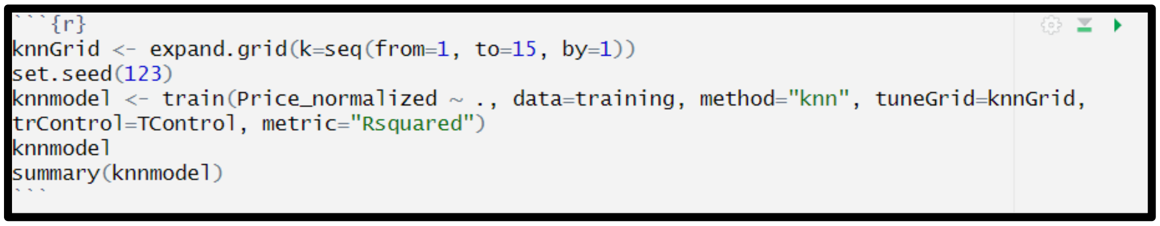


```{r}
prediction.train <- predict(ridge_model, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
ridge.r2 <- 1.0-train.rss/default.train.rss
ridge.r2
prediction.test <- predict(ridge_model, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
ridge.pseudoR2 <- 1.0-test.rss/default.test.rss
ridge.pseudoR2
```

The ridge regression model's performance was evaluated using $R^2$ and pseudo- $R^2$ metrics, which measure how well the model explains variance in both training and test datasets compared to a baseline model predicting only the mean. For training data, $R^2$= 0.6495967, indicating that approximately 65% of variance in ***Price_normalized*** is explained by the model. On test data, pseudo - $R^2$ = 0.6254821, showing similar performance on unseen data and suggesting good generalization ability.

## 3.3 K-NN Model

In the K-nearest neighbors (KNN) model, we leveraged its ability to adapt to non-linear relationships in the data. In our implementation, we performed a grid search to identify the optimal number of neighbors , testing values from 1 to 15 in increments of 1. The model was trained using the ***caret*** package in R, with ***Price normalized*** as the dependent variable and all other features as predictors. Cross-validation was applied using a predefined control object (***TControl)*** to ensure the reliability of the model's results, and the performance metric used was R-squared. After training, we summarized the model's performance for each value of ￼, identifying the optimal value based on R-squared. Additionally, a plot of the model was generated to visually assess the relationship between ￼ and performance metrics, providing further insights into the model's fit and generalization capability.

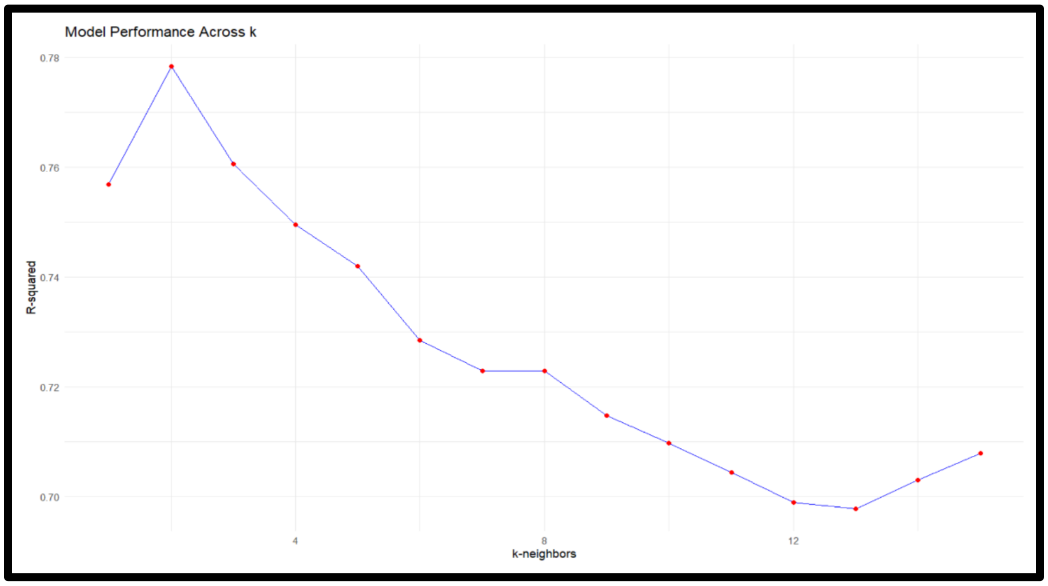


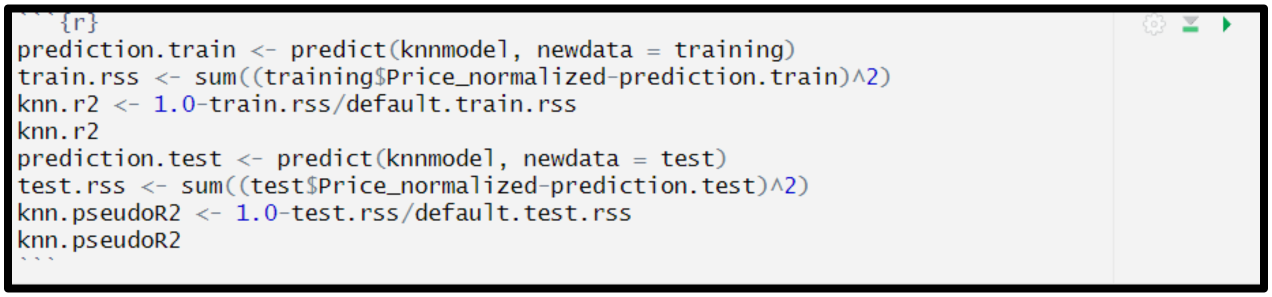
~~~
```{r}
knnGrid <- expand.grid(k=seq(from=1, to=15, by=1))
set.seed(123)
knnmodel <- train(Price_normalized ~ ., data=training, method="knn", tuneGrid=knnGrid,
trControl=TControl, metric="Rsquared")
knnmodel
summary(knnmodel)
```
~~~

The table below shows the tuning process for the hyperparameters to find the best set-up. The resulting k is equal to 2.

| k | RMSE | Rsquared | MAE |
|---|---|---|---|
| 1 | 0.030531 | 0.756866 | 0.014744 |
| **2** | **0.028342** | **0.778301** | **0.015089** |
| 3 | 0.029513 | 0.760594 | 0.015652 |
| 4 | 0.030107 | 0.749499 | 0.015937 |
| 5 | 0.030301 | 0.741971 | 0.016149 |
| 6 | 0.031151 | 0.72847 | 0.01643 |
| 7 | 0.031563 | 0.722915 | 0.016606 |
| 8 | 0.031553 | 0.722877 | 0.016755 |
| 9 | 0.031957 | 0.714727 | 0.01696 |
| 10 | 0.032212 | 0.709747 | 0.017041 |
| 11 | 0.032489 | 0.704379 | 0.017208 |
| 12 | 0.032693 | 0.698937 | 0.017294 |
| 13 | 0.032754 | 0.697782 | 0.017349 |
| 14 | 0.032435 | 0.702971 | 0.017292 |
| 15 | 0.032146 | 0.707841 | 0.017261 |

we are plotting the hyperparameter tuning for the better representation of the process





```{r}
prediction.train <- predict(knnmodel, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
knn.r2 <- 1.0-train.rss/default.train.rss
knn.r2
prediction.test <- predict(knnmodel, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
knn.pseudoR2 <- 1.0-test.rss/default.test.rss
knn.pseudoR2
```

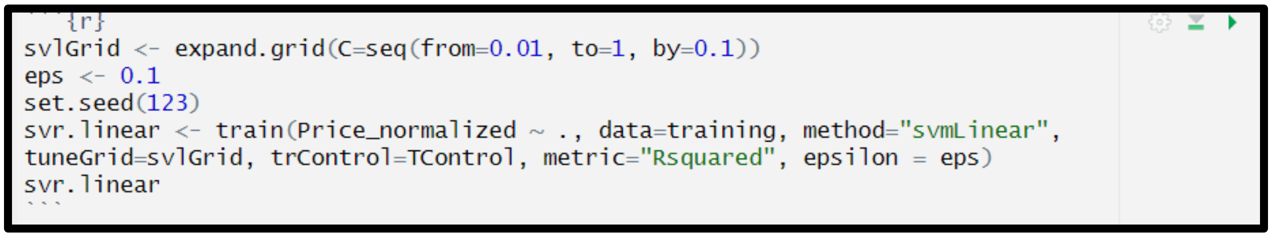
The KNN model's performance was evaluated using $R^2$ and pseudo- $R^2$ metrics, which measure how well the model explains variance in both training and test datasets compared to a baseline model predicting only the mean. For training data, $R^2 = 0.9081$, indicating that approximately 90.81 % of variance in ***Price_normalized*** is explained by the model. On test data, pseudo- $R^2 = 0.7646633$, showing similar performance on unseen data and suggesting good generalization ability.

## 3.4 Support Vector Regression

Support Vector Regression with a linear kernel attempts to fit a linear hyperplane to the data while maintaining a margin of tolerance (epsilon). It creates a linear decision boundary by finding the optimal hyperplane that maximizes the margin between data points.

### 3.4.1 SVR-Linear

**Linear:**

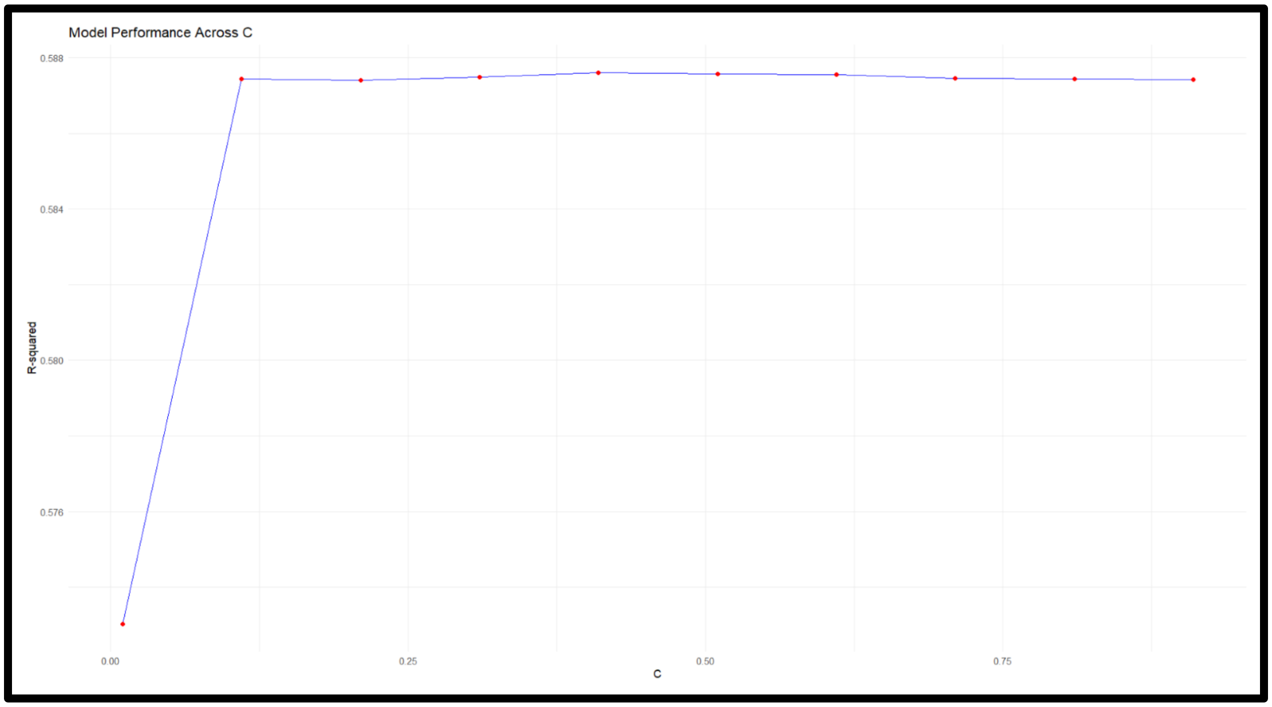


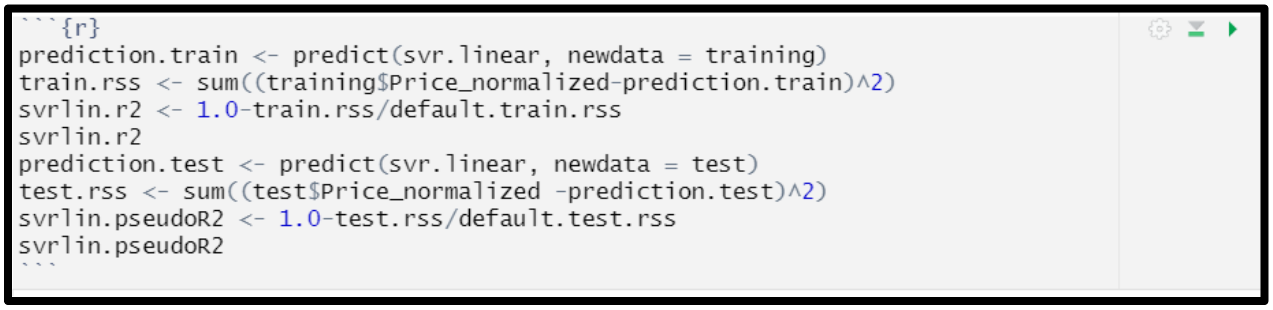
```{r}
svlGrid <- expand.grid(C=seq(from=0.01, to=1, by=0.1))
eps <- 0.1
set.seed(123)
svr.linear <- train(Price_normalized ~ ., data=training, method="svmLinear",
tuneGrid=svlGrid, trControl=TControl, metric="Rsquared", epsilon = eps)
svr.linear
```

The table below shows the tuning process for the hyperparameters to find the best set-up. The resulting c is equal to 0.41.

| C | RMSE | Rsquared | MAE |
|---|---|---|---|
| 0.01 | 0.183028 | 0.573033 | 0.027875 |
| 0.11 | 0.190003 | 0.587442 | 0.027761 |
| 0.21 | 0.19239 | 0.587402 | 0.027869 |
| 0.31 | 0.198185 | 0.587493 | 0.028186 |
| **0.41** | **0.198487** | **0.587606** | **0.028194** |
| 0.51 | 0.199025 | 0.587571 | 0.028222 |
| 0.61 | 0.199234 | 0.587548 | 0.028234 |
| 0.71 | 0.199749 | 0.587456 | 0.028262 |
| 0.81 | 0.200103 | 0.587434 | 0.028282 |
| 0.91 | 0.200117 | 0.587421 | 0.028283 |

we are plotting the hyperparameter tuning for the better representation of the process



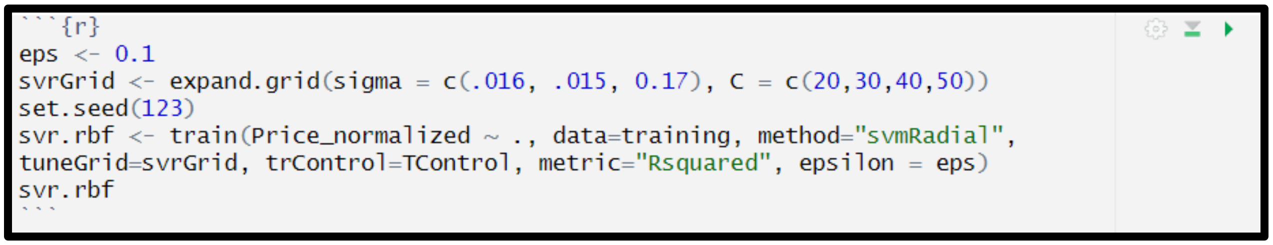


~~~
```{r}
prediction.train <- predict(svr.linear, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
svrlin.r2 <- 1.0-train.rss/default.train.rss
svrlin.r2
prediction.test <- predict(svr.linear, newdata = test)
test.rss <- sum((test$Price_normalized -prediction.test)^2)
svrlin.pseudoR2 <- 1.0-test.rss/default.test.rss
svrlin.pseudoR2
```
~~~

The performance of the SVR (Support Vector Regression) model with a linear kernel was assessed using $R^2$ and pseudo- $R^2$ metrics, which quantify the proportion of variance in the target variable explained by the model compared to a baseline that predicts the mean. For the training dataset, the $R^2$ score was 0.6373, indicating that approximately 63.73% of the variance in ***Price_normalized*** is captured by the model. On the test dataset, the pseudo- $R^2$ score was 0.6126, reflecting a consistent performance on unseen data. These results suggest that the model demonstrates both strong explanatory power and good generalization capability, making it reliable for predicting ***Price_normalized*** across different datasets.

### 3.4.2 SVR-Radial

**Radial:**



~~~
```{r}
eps <- 0.1
svrGrid <- expand.grid(sigma = c(.016, .015, 0.17), C = c(20,30,40,50))
set.seed(123)
svr.rbf <- train(Price_normalized ~ ., data=training, method="svmRadial",
tuneGrid=svrGrid, trControl=TControl, metric="Rsquared", epsilon = eps)
svr.rbf
```
~~~

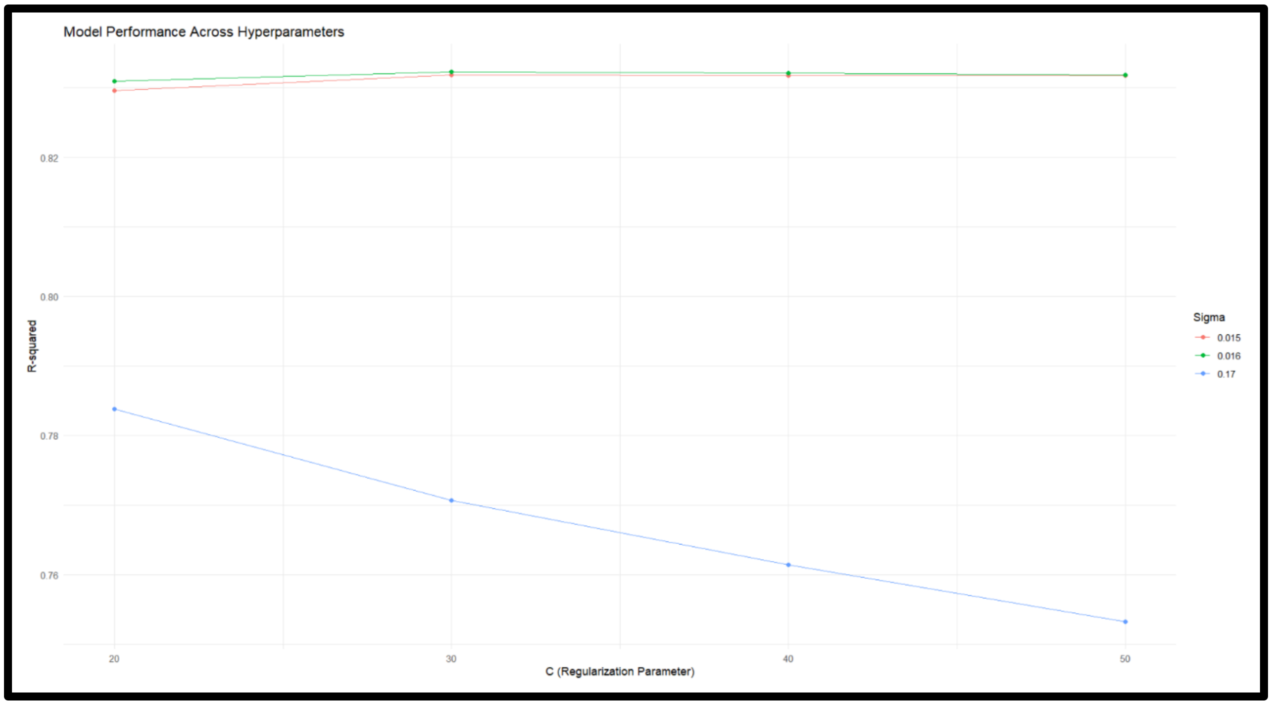
Data Frame of Hyperparameter

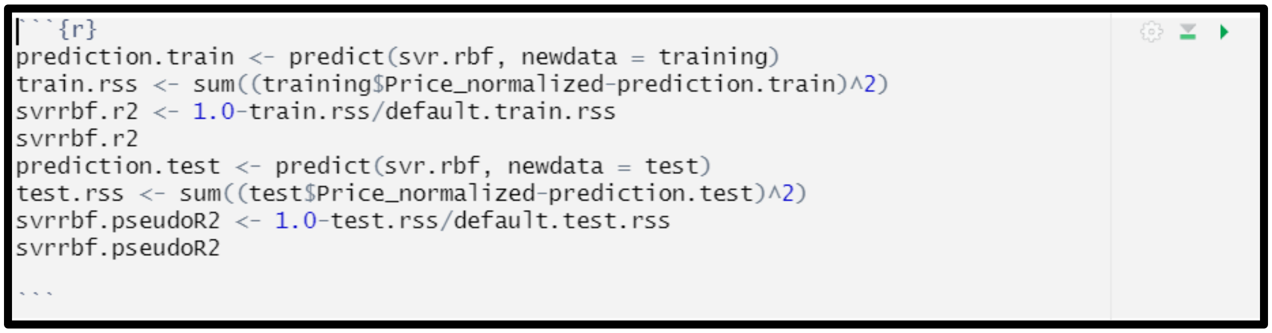
| **Sigma** | **C** |
|---|---|
| 0.15 | 20 |
| 0.16 | 20 |
| 0.17 | 20 |
| 0.15 | 30 |
| 0.16 | 30 |
| 0.17 | 30 |
| 0.15 | 40 |
| 0.16 | 40 |
| 0.17 | 40 |
| 0.15 | 50 |
| 0.16 | 50 |
| 0.17 | 50 |

The table below shows the tuning process for the hyperparameters to find the best set-up. The resulting c is equal to 30 and sigma is equal to 0.16.

| sigma | C | RMSE | Rsquared | MAE |
|---|---|---|---|---|
| 0.015 | 20 | 0.024473 | 0.829589 | 0.013641 |
| 0.015 | 30 | 0.024259 | 0.831789 | 0.013528 |
| 0.015 | 40 | 0.024238 | 0.831716 | 0.013486 |
| 0.015 | 50 | 0.024174 | 0.831738 | 0.01342 |
| 0.016 | 20 | 0.024353 | 0.830959 | 0.013597 |
| **0.016** | **30** | **0.024211** | **0.832304** | **0.01351** |
| 0.016 | 40 | 0.024179 | 0.832084 | 0.013449 |
| 0.016 | 50 | 0.024139 | 0.831846 | 0.013392 |
| 0.17 | 20 | 0.027256 | 0.783798 | 0.014988 |
| 0.17 | 30 | 0.028022 | 0.770636 | 0.015227 |
| 0.17 | 40 | 0.028605 | 0.761445 | 0.015416 |
| 0.17 | 50 | 0.029147 | 0.753261 | 0.015573 |

we are plotting the hyperparameter tuning for the better representation of the process





~~~
```{r}
prediction.train <- predict(svr.rbf, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
svrrbf.r2 <- 1.0-train.rss/default.train.rss
svrrbf.r2
prediction.test <- predict(svr.rbf, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
svrrbf.pseudoR2 <- 1.0-test.rss/default.test.rss
svrrbf.pseudoR2

```
~~~

The svr.linear model's performance was evaluated using $R^2$ and pseudo- $R^2$ metrics, which measure how well the model explains variance in both training and test datasets compared to a baseline model predicting only the mean. For training data, $R^2$ = 0.8613126, indicating that approximately 86.13 % of variance in ***Price_normalized*** is explained by the model. On test data, pseudo- $R^2$ = 0.814688, showing similar performance on unseen data and suggesting good generalization ability.

which drastically improved results, suggesting non-linear relationships between variables.

**Conclusion between SVR and SVL:**

Radial SVR, also known as RBF (Radial Basis Function) kernel SVR, transforms the input space into a higher-dimensional feature space using a Gaussian kernel function where $\gamma$ (gamma) controls the influence of each training example

In our dataset, Radial SVR is particularly well-suited for this Used car selling price prediction dataset for several reasons because of its:

**Complex Feature Relationships**: The dataset contains multiple normalized features (Year, Price, Mileage, Car Age, Price per KM) along with categorical variables (transformed through one-hot encoding), creating complex non-linear relationships between features.

**Price Distribution:** The normalized price values show varying patterns and non-linear relationships with other features, making linear models less effective at capturing these complex interactions.

**Feature Space Transformation**: The RBF kernel's ability to transform the feature space allows it to capture intricate patterns between car attributes and prices, particularly useful when dealing with both numerical and categorical features.

**Flexibility:** The model can adapt to both local and global patterns in the car price data, making it more robust for price prediction across different car segments and conditions.

**Decision Trees:**

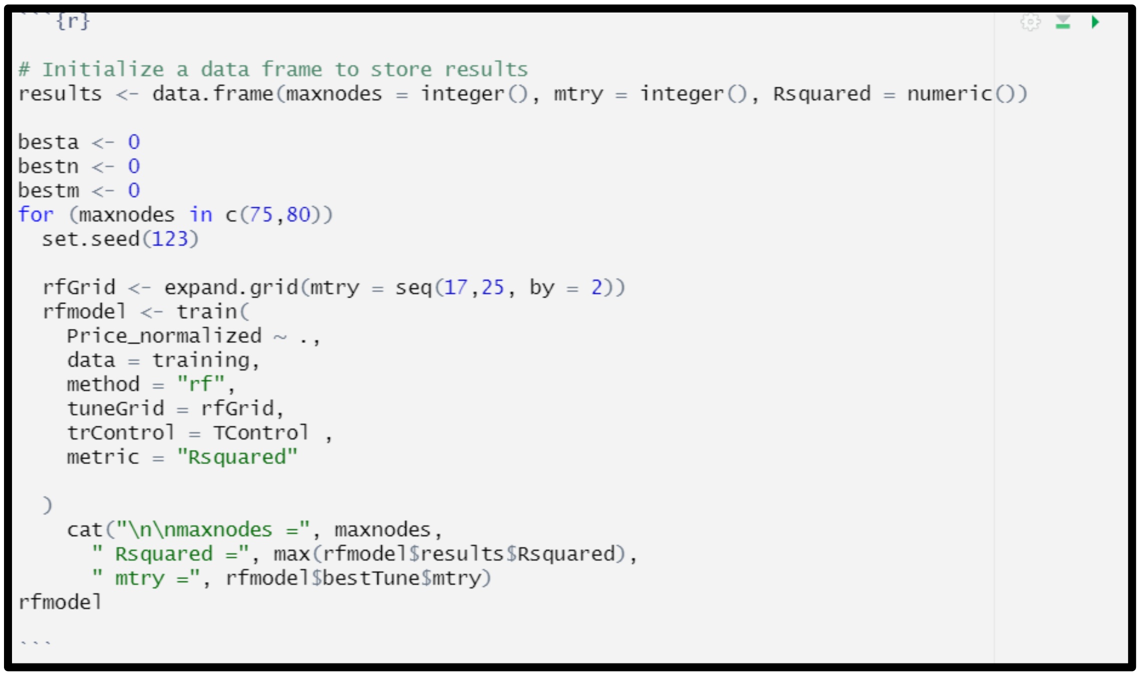
A decision tree recursively partitions the data into subsets based on feature values, creating a tree-like structure of decisions that leads to predicted outcomes. Its hierarchical nature naturally captures non-linear relationships between features (such as ***Year_normalized, Price_normalized, Mileage_normalized***) and handles both numerical and categorical variables (like ***Car_Make, Fuel_Type, Seller_Type***) effectively. The tree's structure provides clear insights into feature importance and decision-making processes, making it particularly valuable for understanding how different car attributes influence price predictions. This interpretability, combined with the ability to handle mixed data types and capture complex interactions, makes decision trees an excellent foundation for more sophisticated ensemble methods. The tree's ability to automatically handle feature interactions and non-linear relationships without requiring explicit feature engineering makes it especially suitable for our dataset, where relationships between features like car age, mileage, and price are inherently non-linear.

The decision trees that we are using in our datasets are:

1. Random Forest
2. Neural Network
3. XG Boost

## 3.5 Random Forest

Random Forest is a versatile ensemble learning algorithm that builds multiple decision trees during training and combines their predictions to improve accuracy and robustness. Each tree in the forest is trained on a random subset of the data, and the algorithm averages their outputs (for regression).

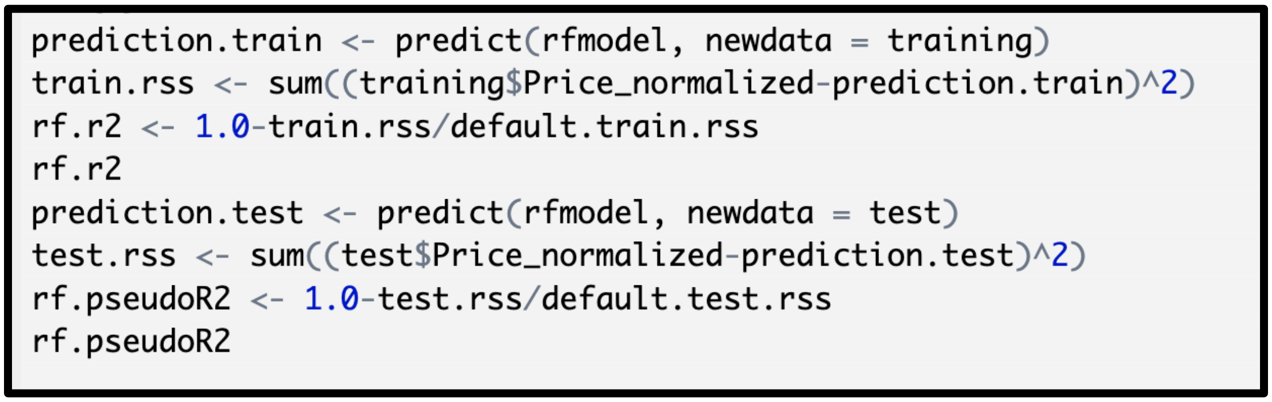


~~~
```{r}

# Initialize a data frame to store results
results <- data.frame(maxnodes = integer(), mtry = integer(), Rsquared = numeric())

besta <- 0
bestn <- 0
bestm <- 0
for (maxnodes in c(75,80))
  set.seed(123)

  rfGrid <- expand.grid(mtry = seq(17,25, by = 2))
  rfmodel <- train(
    Price_normalized ~ .,
    data = training,
    method = "rf",
    tuneGrid = rfGrid,
    trControl = TControl ,
    metric = "Rsquared"

  )
    cat("\n\nmaxnodes =", maxnodes,
      " Rsquared =", max(rfmodel$results$Rsquared),
      " mtry =", rfmodel$bestTune$mtry)
rfmodel

```
~~~

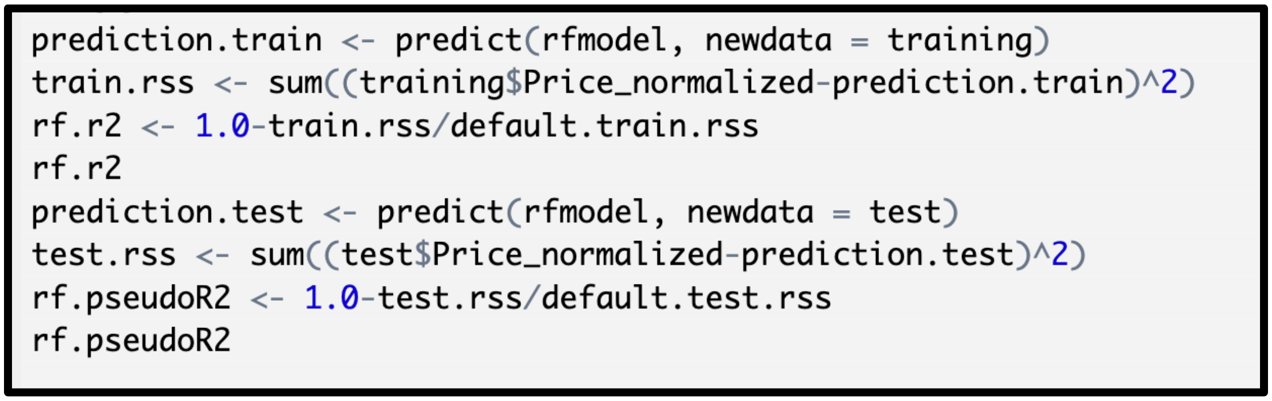
We built and optimized a Random Forest model to predict normalized car prices (Price_normalized). To fine-tune the hyperparameters, we performed a systematic grid search. We first iterated over different values of maxnodes (such as 75 and 80) to control the maximum number of terminal nodes in the decision trees, thereby managing the complexity of individual trees. Additionally, we tuned mtry, the number of features considered at each split, over a range of values from 17 to 25 in increments of 2.

The table below shows the tuning process for the hyperparameters to find the best set-up. The resulting mtry is equal to 25.

| mtry | RMSE | Rsquared | MAE |
|---|---|---|---|
| 17 | 0.01620 | 0.92583 | 0.00470 |
| 19 | 0.01607 | 0.92626 | 0.00439 |
| 21 | 0.01582 | 0.92846 | 0.00412 |
| 23 | 0.01560 | 0.93002 | 0.00388 |
| **25** | **0.01545** | **0.93107** | **0.00369** |



```
prediction.train <- predict(rfmodel, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
rf.r2 <- 1.0-train.rss/default.train.rss
rf.r2
prediction.test <- predict(rfmodel, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
rf.pseudoR2 <- 1.0-test.rss/default.test.rss
rf.pseudoR2
```



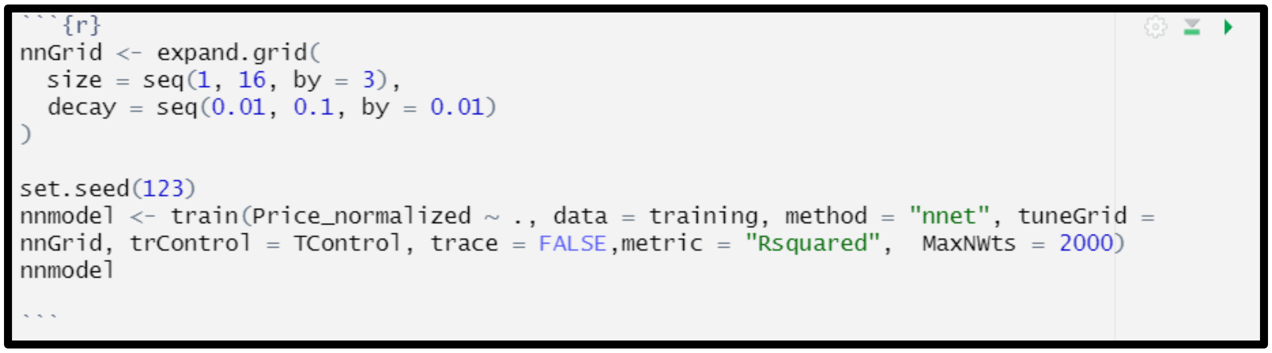
```
prediction.train <- predict(rfmodel, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
rf.r2 <- 1.0-train.rss/default.train.rss
rf.r2
prediction.test <- predict(rfmodel, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
rf.pseudoR2 <- 1.0-test.rss/default.test.rss
rf.pseudoR2
```

The model was evaluated using $R^2$, which measures the proportion of variance in the target variable explained by the model, with higher values indicating better performance. The best configuration achieved an $R^2$ of 98.5% on the training data and 86.3% on the test data, demonstrating strong predictive accuracy and good generalization to unseen data.

This approach allowed us to balance the model's complexity and interpretability while ensuring it performs well on both training and test datasets. The iterative tuning process helped identify the optimal combination of hyperparameters, ensuring that the Random Forest model was both efficient and effective in capturing the relationships within the dataset.

## 3.6 Neural Network

We used a neural network model to predict the normalized selling price of used cars. The model was implemented using the `***caret***` package in R with the `***nnet***` method. To optimize the model's performance, we defined a hyperparameter grid (`***nnGrid***`) that tested different combinations of the number of hidden neurons (`***size***`) ranging from 1 to 30 in steps of 3, and the regularization parameter (`***decay***`) ranging from 0.01 to 0.1 in increments of 0.01. We used the `***train***` function to fit the model, specifying `***Price_normalized***` as the dependent variable and all other features as predictors. Cross-validation settings (`***TControl***`) were applied to ensure robust evaluation, and the model was tuned using R-squared as the performance metric. The `***trace***` parameter was set to `***FALSE***` to suppress output during training, and the maximum number of weights (`***MaxNWts***`) was set to 2000 to handle larger models. The final model was summarized to identify the best combination of hyperparameters that provided the highest R-squared value. This approach allowed us to systematically train and optimize the neural network model for accurate predictions.



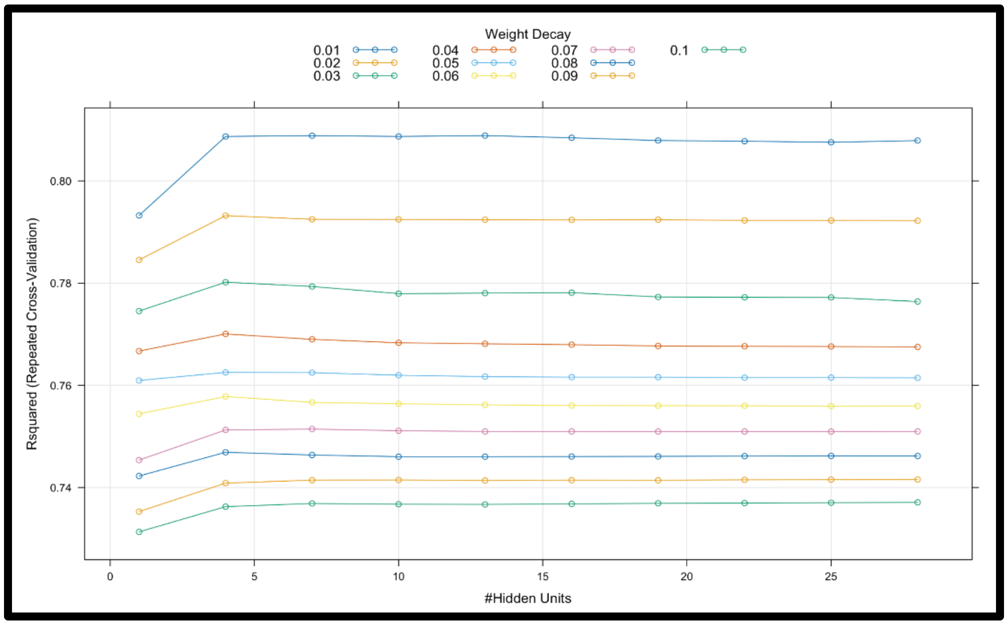
```{r}
nnGrid <- expand.grid(
  size = seq(1, 16, by = 3),
  decay = seq(0.01, 0.1, by = 0.01)
)

set.seed(123)
nnmodel <- train(Price_normalized ~ ., data = training, method = "nnet", tuneGrid =
nnGrid, trControl = TControl, trace = FALSE,metric = "Rsquared",  MaxNWts = 2000)
nnmodel

```

The table below shows the tuning process for the hyperparameters only with decay 0.01, to find the best set-up.

| size | decay | RMSE | Rsquared | MAE |
|---|---|---|---|---|
| 1 | 0.01 | 0.027168 | 0.793242 | 0.017304 |
| 4 | 0.01 | 0.026123 | 0.808698 | 0.015864 |
| 7 | 0.01 | 0.0261 | 0.808858 | 0.015811 |
| 10 | 0.01 | 0.026118 | 0.808699 | 0.015822 |
| **13** | **0.01** | **0.026111** | **0.808868** | **0.015823** |
| 16 | 0.01 | 0.026148 | 0.808442 | 0.015848 |

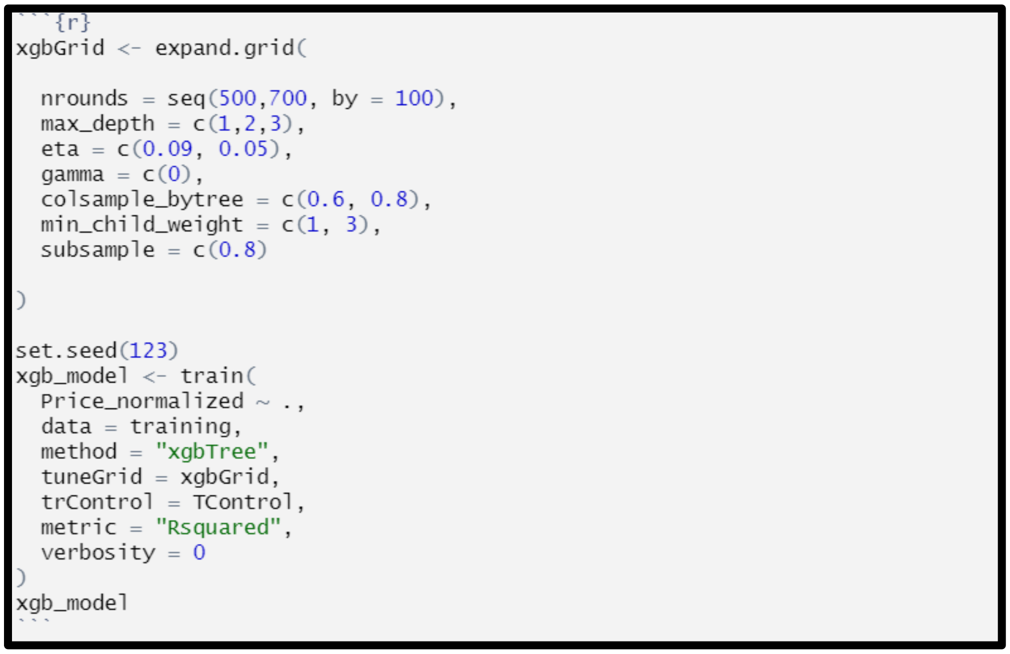
The optimal neural network model was selected based on the largest R-squared value, which evaluates how well the model explains the variance in the target variable. After testing various combinations of hyperparameters, the best configuration was found to have 13 neurons in the hidden layer (size = 13) and a regularization parameter (decay) of 0.01. This combination provided the highest accuracy during training, making it the final model for predicting the normalized selling price of used cars.

We get an accuracy of 82.5% and 78.7% for training and test sets, respectively. We are plotting the hyperparameter tuning for the better representation of the process



## 3.7 XG Boost

XGBoost (eXtreme Gradient Boosting) is a powerful machine learning algorithm that excels at handling complex datasets with mixed feature types, making it particularly well-suited for this car price prediction dataset. The algorithm's effectiveness stems from its ability to capture intricate relationships between various car attributes, including normalized numerical features (Year, Price, Mileage, Car Age) and categorical variables encoded through one-hot encoding (Car Make, Fuel Type, Seller Type). XGBoost builds an ensemble of decision trees sequentially, with each tree correcting the errors of its predecessors, while incorporating regularization techniques to prevent overfitting. This approach is especially valuable for the current dataset, where car prices are influenced by multiple interacting factors and non-linear relationships between features. The algorithm's built-in handling of missing values and sparse data from one-hot encoded categorical variables makes it an optimal choice for processing the complex automotive pricing patterns present in the data.



~~~
```{r}
xgbGrid <- expand.grid(

  nrounds = seq(500,700, by = 100),
  max_depth = c(1,2,3),
  eta = c(0.09, 0.05),
  gamma = c(0),
  colsample_bytree = c(0.6, 0.8),
  min_child_weight = c(1, 3),
  subsample = c(0.8)

)

set.seed(123)
xgb_model <- train(
  Price_normalized ~ .,
  data = training,
  method = "xgbTree",
  tuneGrid = xgbGrid,
  trControl = TControl,
  metric = "Rsquared",
  verbosity = 0
)
xgb_model
```
~~~

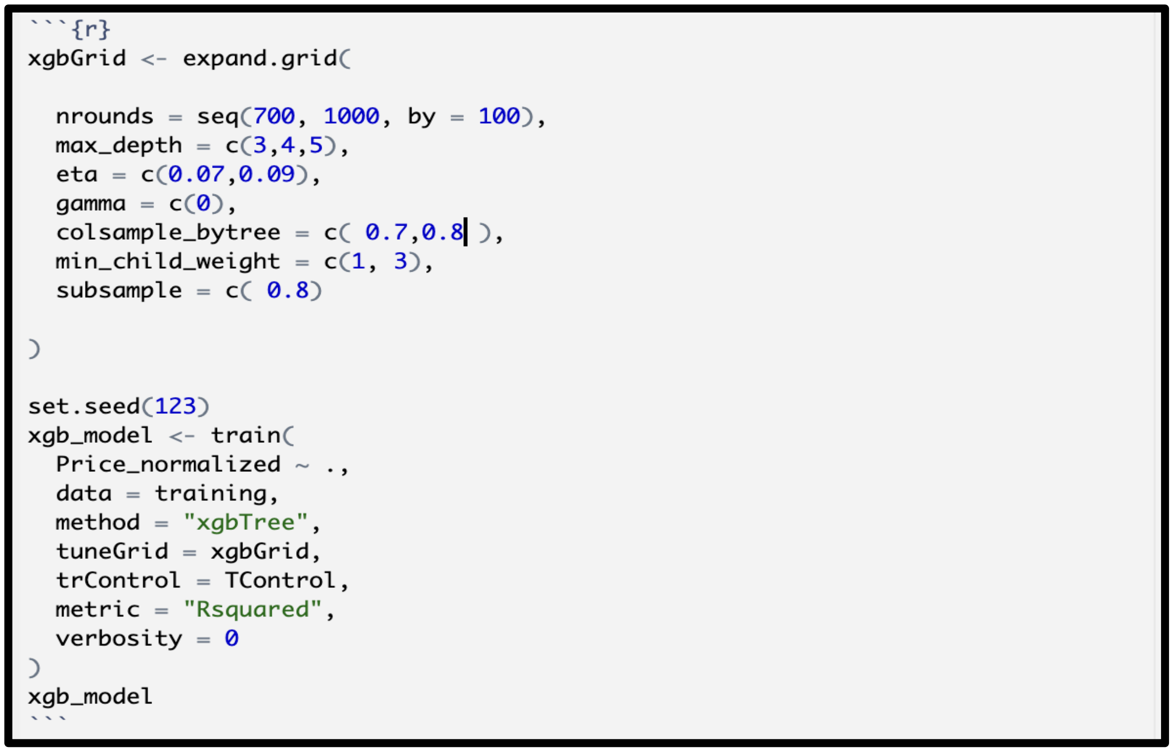
## Key Components of the Code

1. **nrounds (Number of Boosting Iterations):**
   - Specifies the number of trees to build sequentially which is Specified as c(500, 700 by=100).
2. **max_depth:**
   - Controls the maximum depth of each tree in the ensemble, which is Defined as

     c(2, 3, 4)
3. **eta (Learning Rate):**
   - Determines how much each tree contributes to the final prediction, which we have mentioned as c(0.09, 0.1).
4. **gamma:**
   - Adds a penalty for creating new splits in trees, we are mentioning the gamma as c(0) to reduce the computational time

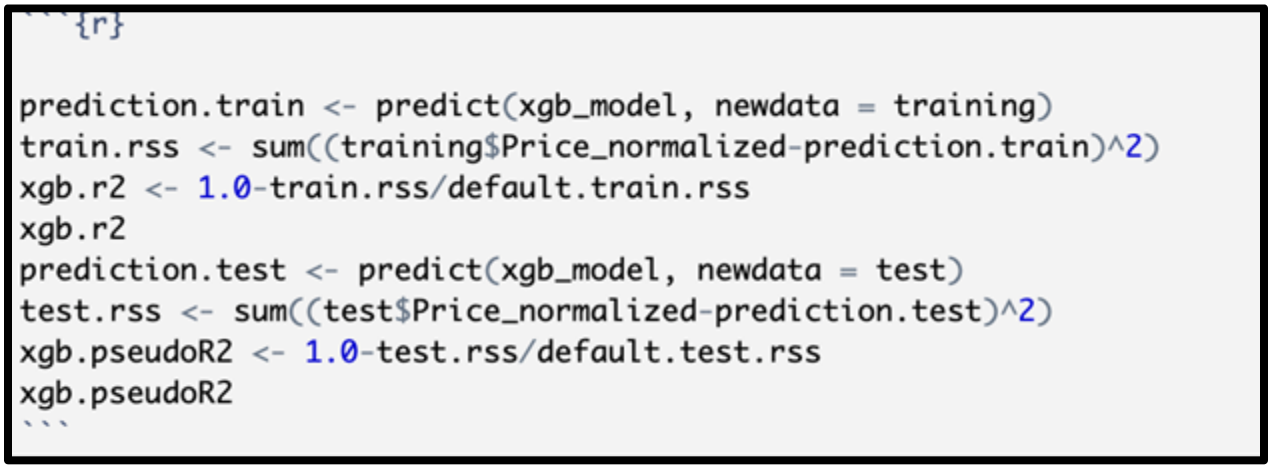
5. **colsample_bytree:**
    - Specifies the fraction of features to randomly sample for each tree, which we c(0.6,0.8)
6. **min_child_weight (Minimum Sum of Instance Weights per Leaf):**
    - Prevents leaves with insufficient data from being created, c(1,3)
    - Suggestion: Add a grid like 1, 3, 5 to test for potential underfitting/overfitting.
7. **subsample:**
    - Specifies the fraction of training data to randomly sample for each tree, c(0.8).

| eta | max_depth | Colsample_bytree | min_child_wight | nrounds | RSME | Rsquared |
|---|---|---|---|---|---|---|
| 0.09 | 3 | 0.6 | 1 | 500 | 0.01364 | 0.9460 |
| 0.09 | 3 | 0.8 | 3 | 600 | 0.01348 | 0.9500 |
| **0.09** | **3** | **0.8** | **3** | **700** | **0.01190** | **0.9586** |



~~~
```{r}
xgbGrid <- expand.grid(

  nrounds = seq(700, 1000, by = 100),
  max_depth = c(3,4,5),
  eta = c(0.07,0.09),
  gamma = c(0),
  colsample_bytree = c( 0.7,0.8 ),
  min_child_weight = c(1, 3),
  subsample = c( 0.8)

)

set.seed(123)
xgb_model <- train(
  Price_normalized ~ .,
  data = training,
  method = "xgbTree",
  tuneGrid = xgbGrid,
  trControl = TControl,
  metric = "Rsquared",
  verbosity = 0
)
xgb_model
```
~~~

| eta | max_depth | Colsample_bytree | min_child_wight | nrounds | RSME | Rsquared |
|---|---|---|---|---|---|---|
| **0.09** | **4** | **0.8** | **3** | **1000** | **0.011163** | **0.963901** |



~~~
```{r}

prediction.train <- predict(xgb_model, newdata = training)
train.rss <- sum((training$Price_normalized-prediction.train)^2)
xgb.r2 <- 1.0-train.rss/default.train.rss
xgb.r2
prediction.test <- predict(xgb_model, newdata = test)
test.rss <- sum((test$Price_normalized-prediction.test)^2)
xgb.pseudoR2 <- 1.0-test.rss/default.test.rss
xgb.pseudoR2
```
~~~

The Neural network model's performance was evaluated using and pseudo- $R^2$ metrics, which measure how well the model explains variance in both training and test datasets compared to a baseline model predicting only the mean. For training data, $R^2 = 0.9997582$, indicating that approximately 99.98 % of variance in ***Price_normalized*** is explained by the model. On test data, pseudo- $R^2 = 0.9284257$, showing similar performance on unseen data and suggesting good generalization ability.

# 4. RESULTS AND CONCLUSION

**The comparison of all the experiments is shown in the table below:**

| SR# | Model | Accuracy (Training)% | Accuracy (Test) % | MAE | RMSE |
|---|---|---|---|---|---|
| 1 | Linear Regression | 67.33 | 67.23 | 0.0339 | 0.2732 |
| 2 | Ridge Regression | 64.95 | 62.54 | 0.0341 | 0.2694 |
| 3 | K-NN | 90.81 | 76.46 | 0.0150 | 0.0283 |
| 4 | SVR Linear | 63.73 | 61.26 | 0.0281 | 0.1984 |
| 5 | SVR Radial | 86.13 | 81.46 | 0.0135 | 0.0242 |
| 6 | Random Forest | 98.5 | 86.3 | 0.0036 | 0.0154 |
| 7 | Neural Network | 82.5 | 78.7 | 0.0158 | 0.0261 |
| 8 | XG Boost | 99.98 | 92.84 | 0.0027 | 0.0111 |

## Best Model in the Table

The **XGBoost** model is the best-performing model in the table based on the following criteria:

1. **Accuracy**: XGBoost achieved the highest accuracy on both the training dataset (99.98%) and the test dataset (92.84%).
2. **Error Metrics**: It has the lowest Mean Absolute Error (MAE) of 0.0027 and the lowest Root Mean Squared Error (RMSE) of 0.0111, indicating excellent prediction performance and minimal deviation from actual values.
3. **Generalization**: XGBoost maintains a strong balance between training and test accuracy, indicating its ability to generalize well to unseen data.

## Conclusion

Among all the models evaluated, **XGBoost** stands out as the most effective and reliable model for predicting the target variable. Its superior performance across accuracy and error metrics demonstrates its ability to capture complex relationships in the data while avoiding overfitting. With its robust generalization capability, XGBoost is the optimal choice for deployment in real-world scenarios where both accuracy and reliability are critical. Other models, such as Random Forest and SVR Radial, also perform well, but they fall short of XG Boost's exceptional accuracy and lower error rates.

# BIBLIOGRAPHY

Noor, K., & Jan, S. (2017). Vehicle Price Prediction System using Machine Learning Techniques. *International Journal of Computer Applications*, 27-31.

Kuiper, S. (2008). Introduction to Multiple Regression: How Much Is Your Car Worth? *Journal of Statistics Education*. doi:10.1080/10691898.2008.11889579

Monburinon, N., Chertchom, P., Kaewkiriya, T., Rungpheung, S., Buya, S., & Boonpou, P. (2018). Prediction of Prices for Used Car by Using Regression Models. *5th International Conference on Business and Industrial Research (ICBIR)*, (pp. 115-119). Bangkok.

Pudaruth, S. (2014). Predicting the Price of Used Cars using Machine Learning. *International Journal of Information & Computation Technology*, 754-764.

K.Samruddhi, & Kumar, D. R. (2020, September). Used Car Price Prediction using K-Nearest Neighbor Based Model. *International Journal of Innovative Research in Applied Sciences and Engineering (IJIRASE), 4*(3), 686-689.

Gongqi, S., Yansong, W., & Qiang, Z. (2011). A New Model for Residual Value Prediction of the Used Car Based on BP Neural. *Third International Conference on Measuring Technology and Mechatronics Automation* (pp. 682-685). Shanghai: IEEE. doi:10.1109/ICMTMA.2011.455

Jian Da Wu, C.-c. H.-C. (2017). "An expert system of price forecasting for used cars using adaptive. *ELSEVEIR, 16*, 417-957.

Listiani, M. (2009). Support Vector Regression Analysis for Price Prediction in a Car Leasing Application. *Master Thesis*. Hamburg: Hamburg University of Technology.