Title: Seoul Bike Sharing Demand Prediction

Abstract:

Bicycle sharing systems have become an important part of transportation in many cities around the world due to their positive impact on health, traffic congestion, and the environment. With the help of intelligent transportation systems and information technology, these systems have seen enormous growth over the past few decades. This study discusses the Ddareungi bike sharing system in South Korea and proposes a data mining-based approach, including weather data, to predict the demand for rental bikes. The study applies various techniques, such as exploratory data analysis, linear regression, decision tree regressor, and random forest regressor, using the R programming language. Overall, the paper highlights the importance of managing bike rental demand to provide continuous and convenient service for users.

Bakht Singh Basaram – 11546730
Sahithi Chindam – 11617204
Santhoshi Indrakanti – 11610865
Pavan Kumar Suthari – 11550894

Table of Contents

Problem statement

The business understanding of this problem is to provide a stable supply of rental bikes for public use in urban cities. The availability and accessibility of rental bikes at the right time can enhance mobility comfort and reduce waiting time. To achieve this, it is important to predict the number of bikes rented at each hour. This prediction can help in managing the supply of rental bikes and ensure that there are enough bikes available for public use. The dataset contains weather information that can affect bike rentals, so it is essential to consider these factors when predicting the number of bikes rented. The best regression model can be used to forecast the demand for bikes and plan the supply accordingly. Overall, the aim is to provide a reliable and convenient rental bike service to the public, contributing to the sustainable development of urban cities.

Data Specification

The Seoul Bike Sharing Demand dataset contains weather-related features such as temperature, humidity, wind speed, visibility, dew point, solar radiation, snowfall, and rainfall. These features are used to predict the demand for bike rentals in Seoul, South Korea. There are also categorical features such as season, holiday, and functioning day. The season column has four categories, namely spring, summer, autumn, and winter, while the holiday column has two categories, holiday, and no holiday. The functioning day column indicates whether the day is a working day or not. The output variable, rented bike count, represents the number of bikes rented per hour. The dataset contains 8760 records and 14 columns with no null values, making it suitable for a regression problem. This dataset contains hourly data related to bike rentals in Seoul, South Korea, between 1st of January 2017 and 30th of November 2018. The variables included in the dataset are described below:

Features of Dataset:

1. **Date:** The date on which the data was recorded, in the format yyyy-mm-dd. This variable provides information about when the bike rental activity occurred and allows for the identification of patterns and trends over time.

2. **Rented Bike Count:** The total number of bikes rented in that hour. This variable is the target variable for most analyses and is used to predict bike rental demand based on other factors.
3. **Hour:** The hour of the day for which the data was recorded, in 24-hour format. This variable provides information about the time of day when bike rentals are most in demand.
4. **Temperature:** The temperature in Celsius at the time of recording. Temperature can influence bike rental demand, with higher temperatures generally leading to higher demand.
5. **Humidity:** The relative humidity at the time of recording. Humidity can also influence bike rental demand, with high humidity potentially reducing demand.
6. **Wind Speed:** The wind speed in meters per second at the time of recording. Wind speed can affect how comfortable it is to ride a bike, and high wind speeds may reduce bike rental demand.
7. **Visibility:** The visibility in meters at the time of recording. Poor visibility due to fog or other weather conditions may reduce bike rental demand.
8. **Dew point Temperature:** The dew point temperature in Celsius at the time of recording. Dew point temperature is a measure of how much moisture is in the air and can affect how comfortable it is to be outside.
9. **Solar Radiation:** The solar radiation in Mega Joule per square meter at the time of recording. Solar radiation can influence temperature and may also affect how comfortable it is to be outside.
10. **Rainfall:** The amount of rainfall in mm at the time of recording. Rainfall can significantly reduce bike rental demand.
11. **Snowfall:** The amount of snowfall in cm at the time of recording. Snowfall can make cycling difficult or impossible, and therefore greatly reduce bike rental demand.
12. **Seasons:** The season of the year for which the data was recorded. The seasons are categorized as follows: Spring, Summer, Autumn, and Winter. Seasonality is an important factor to consider when analyzing bike rental demand, as it can affect demand in different ways.

13. **Holiday:** A binary variable indicating whether the day was a holiday. Holidays can affect bike rental demand, with some holidays leading to increased demand while others may lead to reduced demand.
14. **Functioning Day:** A binary variable indicating whether the day was a functioning day (i.e., not a weekend or holiday). This variable can be used to distinguish between days when people are more likely to use bikes for transportation versus days when they are more likely to use them for leisure activities.

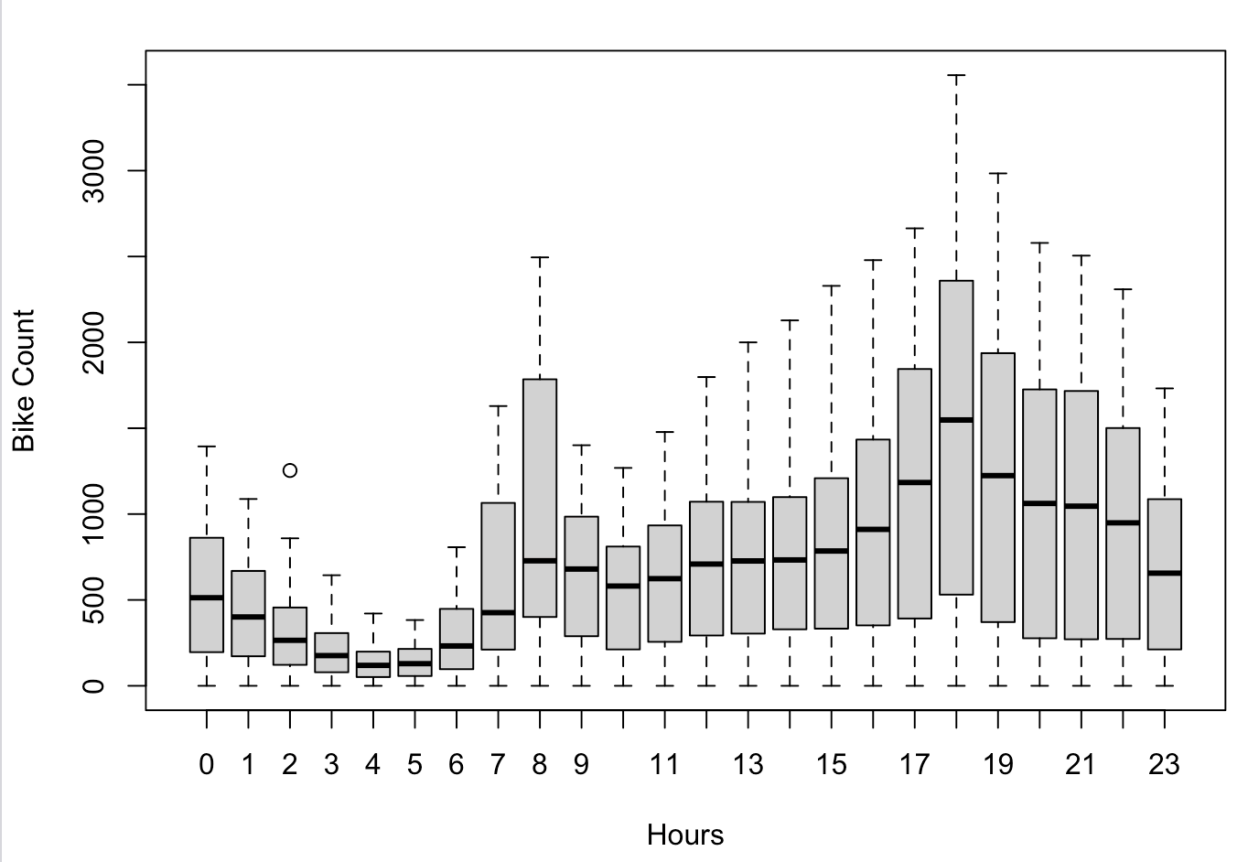
```
> seoul_bike_data<-read.csv("SeoulBikeData.csv",fileEncoding = "Latin1",check.names = F)
> # get the first glimpse of the data
> glimpse(seoul_bike_data)
Rows: 8,760
Columns: 14
$ Date                       <chr> "01/12/2017", "01/12/2017", "01/12/2017", "01/12/2017…
$ `Rented Bike Count`        <int> 254, 204, 173, 107, 78, 100, 181, 460, 930, 490, 339,…
$ Hour                       <int> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,…
$ `Temperature(°C)`          <dbl> -5.2, -5.5, -6.0, -6.2, -6.0, -6.4, -6.6, -7.4, -7.6,…
$ `Humidity(%)`              <int> 37, 38, 39, 40, 36, 37, 35, 38, 37, 27, 24, 21, 23, 2…
$ `Wind speed (m/s)`         <dbl> 2.2, 0.8, 1.0, 0.9, 2.3, 1.5, 1.3, 0.9, 1.1, 0.5, 1.2…
$ `Visibility (10m)`         <int> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000,…
$ `Dew point temperature(°C)` <dbl> -17.6, -17.6, -17.7, -17.6, -18.6, -18.7, -19.5, -19.…
$ `Solar Radiation (MJ/m2)`  <dbl> 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.01,…
$ `Rainfall(mm)`             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ `Snowfall (cm)`            <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ Seasons                    <chr> "Winter", "Winter", "Winter", "Winter", "Winter", "Wi…
$ Holiday                    <chr> "No Holiday", "No Holiday", "No Holiday", "No Holiday…
$ `Functioning Day`          <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Yes…
> # get dimensions of the data
> dim(seoul_bike_data)
[1] 8760   14
> # Remove the brackets and their contents from column names
> colnames(seoul_bike_data) <- gsub("\\(.*?\\)", "", colnames(seoul_bike_data))
> #removing 1st column or feature i.e "Date"
> seoul_bike_data <- seoul_bike_data[, c(-1)]
> # print all the column names after removing "Date" feature
> names(seoul_bike_data)
 [1] "Rented Bike Count"     "Hour"                  "Temperature"
 [4] "Humidity"              "Wind speed "           "Visibility "
 [7] "Dew point temperature" "Solar Radiation "      "Rainfall"
[10] "Snowfall "             "Seasons"               "Holiday"
[13] "Functioning Day"
> # change the column names
> colnames(seoul_bike_data)[5] = "Wind speed"
> colnames(seoul_bike_data)[6] = "Visibility"
> colnames(seoul_bike_data)[8] = "Solar Radiation"
> colnames(seoul_bike_data)[10] = "Snowfall"
> colnames(seoul_bike_data) = gsub(" ","_", colnames(seoul_bike_data))
> names(seoul_bike_data)
 [1] "Rented_Bike_Count"     "Hour"                  "Temperature"
 [4] "Humidity"              "Wind_speed"            "Visibility"
 [7] "Dew_point_temperature" "Solar_Radiation"       "Rainfall"
[10] "Snowfall"              "Seasons"               "Holiday"
[13] "Functioning_Day"
```

- In our dataset, we have 8760 rows and 14 columns.

- We have removed special characters in the column names. And also the trailing spaces.
- This is important because special characters and trailing spaces can cause errors when trying to manipulate or analyse the data.

```
> # plotting a scatter plot between "Hour" and "Rented Bike Count" feature
> boxplot(seoul_bike_data$Rented_Bike_Count ~ seoul_bike_data$Hour, xlab = "Hours", ylab="Bike C
ount")
> # converting "Hour" column into "Categorical Column" based on the above plot
> seoul_bike_data$Hour <- ifelse(seoul_bike_data$Hour < 7, "low",
+                         ifelse(seoul_bike_data$Hour >= 9 & seoul_bike_data$Hour <=15, "Averag
e", "High"))
> # looking for null values in the data
> colSums(is.na(seoul_bike_data)) %>% sort()
    Rented_Bike_Count                 Hour          Temperature             Humidity
                    0                    0                    0                    0
         Wind_speed           Visibility Dew_point_temperature       Solar_Radiation
                    0                    0                    0                    0
             Rainfall             Snowfall              Seasons              Holiday
                    0                    0                    0                    0
      Functioning_Day
                    0
```

- We have printed sum of null values in each column. There are no null values in the dataset.
- This is important because null values, also known as missing values, can cause issues when analysing or modelling data. Depending on the analysis or model, null values may need to be handled in a specific way, such as filling them in with a default value or removing the entire row.
- Knowing that there are no null values in the dataset allows for more accurate and efficient analysis or modelling.
- Before implementing any feature selection techniques, it is recommended to first perform a quick check of the entire dataset to identify any missing values in any of the columns. This step is important to ensure the integrity of the data and to avoid potential errors in subsequent analyses.
- After performing a check of the dataset, we confirmed that there are no missing values in any of the columns, allowing us to proceed with our analysis.



The boxplot analysis reveals that the number of rented bikes varies significantly according to the hour of the day. Specifically, the highest number of rentals occurs during the morning rush hour from 7 a.m. to 9 a.m., and there is a steady level of rentals during the midday hours until 3 p.m. Thereafter, there is a gradual increase in rentals until the evening hours. To simplify the interpretation, we categorized the hours into three groups based on the observed trends: "Low" for hours with less than 7 rentals, "Average" for hours with between 10 to 15 rentals, and "High" for hours with high rental activity (7-9 a.m. and 4-10 p.m.). This approach provides a more intuitive understanding of the data, compared to using raw numbers.

As our dataset includes both categorical and numerical features, we first separated them into two separate objects. We created a list of all categorical variables, converted them from character to factor variables, and saved them as factors. We also created a list of all numeric features, which we stored in a vector named "numerical". This process will allow us to perform further analyses on these distinct types of features.

```
> # copying categorical columns
> bikes_categorical <- sapply(seoul_bike_data, is.character) %>% which() %>% names()
> #converting all categorical variables to factors by applying a loop
> for (variable in bikes_categorical) {
+   seoul_bike_data[[variable]] <- as.factor(seoul_bike_data[[variable]])
+ }
> #list of numerical features in a separate vector
> bikes_numerical <- sapply(seoul_bike_data, is.numeric) %>% which () %>% names()
```

Code explanation:

- It identifies the categorical columns in the dataset 'seoul_bike_data' using the 'sapply()' function with the 'is.character()' function as an argument. The resulting indices are extracted using which() and converted to column names using names(). These column names are stored in the variable bikes_categorical.
- For each variable, the code converts it to a factor using the 'as.factor()' function and replaces the original column in 'seoul_bike_data' with the new factor column.
- Similarly, a vector of column names for numerical features is created using sapply() function with the is.numeric() function as an argument. The resulting indices are extracted using which() and converted to column names using names(). These column names are stored in the variable bikes_numerical.

Data Partitioning

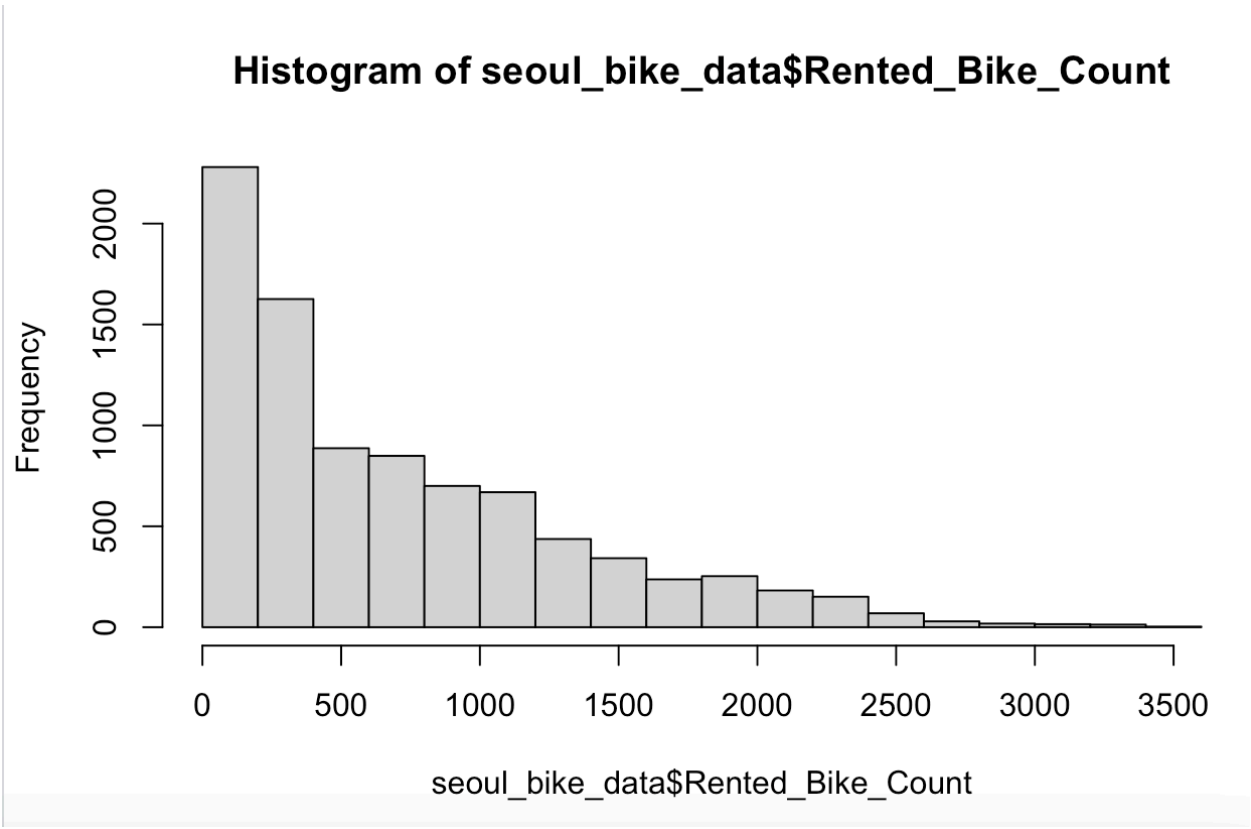
To prepare the data for analysis, we divided it into two separate samples - a training sample and a test sample. To do this, we used the "createDataPartition()" function from the caret package, which selects 70% of the observations and returns a vector of their indexes. We then applied this vector to divide the data into a training set and a test set.

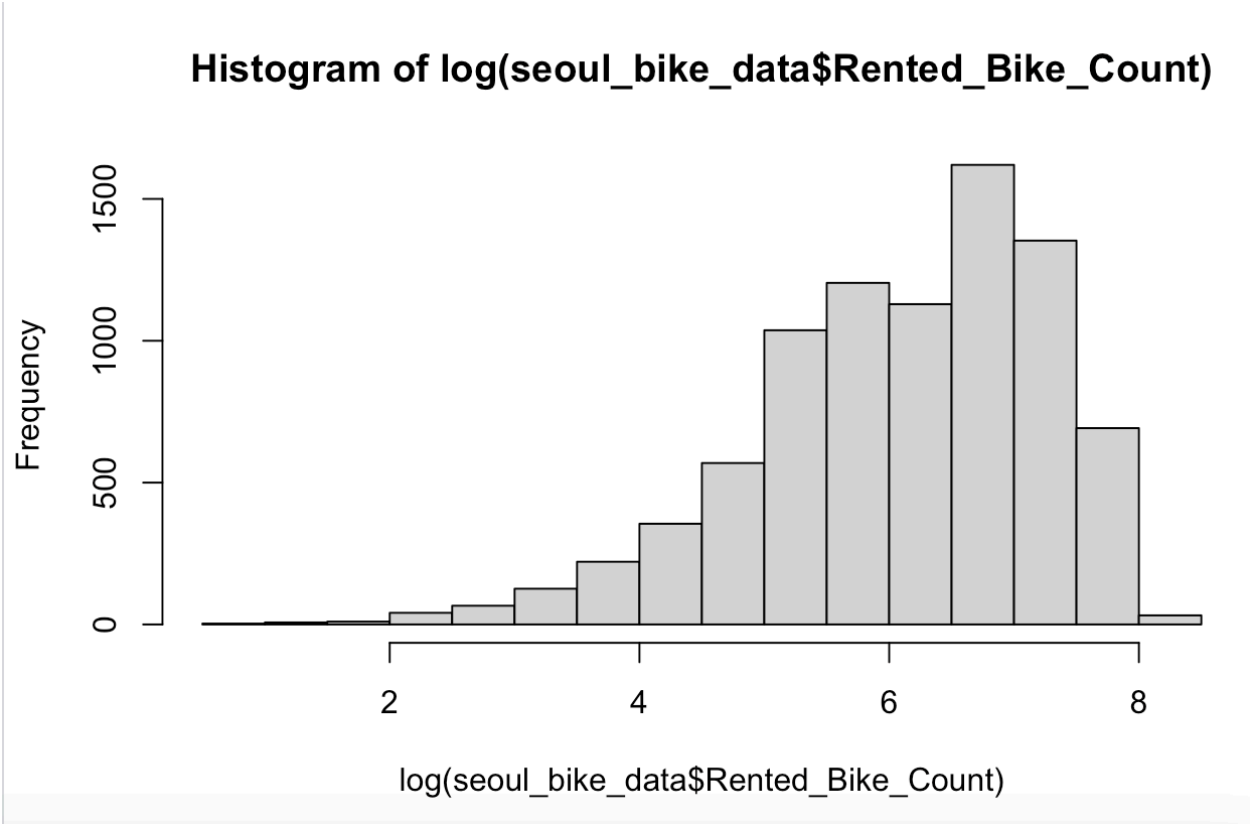
We also checked the frequency distribution of the target variable, which in this case is the number of rented bikes (bikes). While the maximum values of the distribution differ, the other statistics are quite similar. It is important to note that any transformations or adjustments made to the variable distributions should be applied only to the training data. Therefore, we also examined the distribution of the bike's variable in the training data to ensure that it follows a normal distribution, which is essential for the validity of many statistical analyses.

Code Explanation of Correlation between Numerical Features and Target Variable:

- ‘cor()’ is a function in R that computes the correlation between variables.
- The first argument to ‘cor()’ is the data that you want to compute the correlations for. In this case, it's ‘train_data[,bikes_numerical]’, which selects only the numerical columns of the dataset.
- use = "pairwise.complete.obs" is an argument to cor() that specifies how to handle missing data. In this case, it computes the correlation only for pairs of observations that have complete data (i.e., no missing values).
- %>% is the pipe operator in R, which passes the output of the previous function as the first argument to the next function. In this case, it's passing the output of cor() to the round() function.
- round(digits = 2) is a function that rounds the output of the previous function to two decimal places.
- The output of the code is a correlation matrix that shows the pairwise correlations between the numerical features and the target variable in train_data. The values in the matrix range from -1 to 1, where -1 indicates a perfect negative correlation, 1 indicates a perfect positive correlation, and 0 indicates no correlation.

```
> # to reproduce the same partition
> set.seed(123)
> # using "caret" package to partition the data
> data_which_train <- createDataPartition(seoul_bike_data$Rented_Bike_Count, # target variable
+                                         # percentage of the training sample
+                                         p = 0.7,
+                                         # should result be a list?
+                                         list = FALSE)
> train_data <- seoul_bike_data[data_which_train, ]
> test_data <- seoul_bike_data[-data_which_train, ]
> # looking summary of target column in both training and testing set
> summary(train_data$Rented_Bike_Count)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0     191     504     704    1064    3380
> summary(test_data$Rented_Bike_Count)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0     191     505     706    1066    3556
> # look how data is distributed in target feature
> hist(seoul_bike_data$Rented_Bike_Count)
> hist(log(seoul_bike_data$Rented_Bike_Count))
> # plot correlation between target feature and numerical features
> bikes_correlation <-
+   cor(train_data[,bikes_numerical],
+       use = "pairwise.complete.obs")
> bikes_correlation %>% round(digits = 2)
                     Rented_Bike_Count Temperature Humidity Wind_speed Visibility
Rented_Bike_Count                 1.00        0.54    -0.21       0.12       0.21
Temperature                       0.54        1.00     0.14      -0.03       0.04
Humidity                         -0.21        0.14     1.00      -0.34      -0.55
Wind_speed                        0.12       -0.03    -0.34       1.00       0.18
Visibility                        0.21        0.04    -0.55       0.18       1.00
Dew_point_temperature             0.38        0.91     0.53      -0.18      -0.18
Solar_Radiation                   0.26        0.36    -0.46       0.33       0.15
Rainfall                         -0.12        0.05     0.24      -0.02      -0.17
Snowfall                         -0.14       -0.22     0.11      -0.02      -0.13
                     Dew_point_temperature Solar_Radiation Rainfall Snowfall
Rented_Bike_Count                     0.38            0.26    -0.12    -0.14
Temperature                           0.91            0.36     0.05    -0.22
Humidity                              0.53           -0.46     0.24     0.11
Wind_speed                           -0.18            0.33    -0.02    -0.02
Visibility                           -0.18            0.15    -0.17    -0.13
Dew_point_temperature                 1.00            0.10     0.13    -0.15
Solar_Radiation                       0.10            1.00    -0.08    -0.07
Rainfall                              0.13           -0.08     1.00    -0.02
Snowfall                             -0.15           -0.07    -0.02     1.00
```


Histogram of seoul_bike_data$Rented_Bike_Count
Frequency
0
500
1000
1500
2000
0
500
1000
1500
2000
2500
3000
3500
seoul_bike_data$Rented_Bike_Count


Histogram of log(seoul_bike_data$Rented_Bike_Count)
Frequency
0
500
1000
1500
2
4
6
8
log(seoul_bike_data$Rented_Bike_Count)

Upon examining the histogram of the target variable, we observed that it is not normally distributed. This is a concern for linear regression models, as they assume normality in the data. To address this issue, we applied a log transformation to the variable, which improved the distribution but still resulted in a left-skewed data. Therefore, for linear regression purposes, we will use the logarithmic values of the variable and then inverse the log to compare the RMSE values of different models. This process will help ensure that our model is as accurate as possible.

Initial Feature Selection:

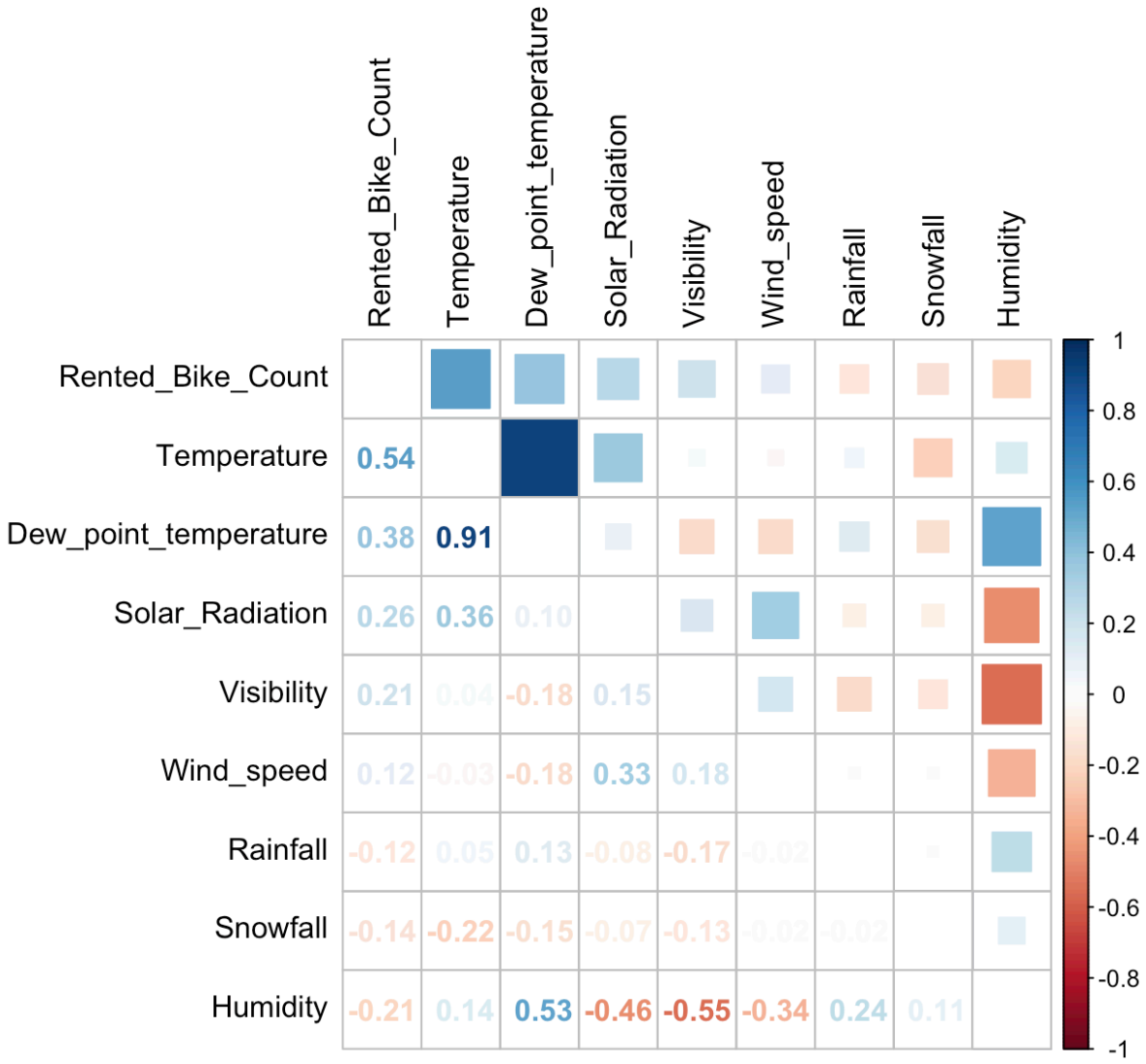
Feature selection is a crucial step in machine learning, as including redundant or weakly correlated variables can reduce the accuracy of our model. Therefore, it is recommended to apply various feature selection techniques to identify and remove these variables. To begin, we examined the relationship between predictors and the outcome variable. Additionally, we checked for collinearity among the predictors to ensure they are not strongly correlated with each other. Since our dataset contains both categorical and numerical features, we performed these analyses separately for each type of feature.

To avoid including redundant and weakly correlated features in our machine learning model, it is important to perform feature selection. This step is crucial in improving model accuracy for both classification and regression tasks. In our analysis, we start by examining the relationship between our predictors and the outcome variable. Additionally, we must ensure that there is no strong collinearity among our predictors. Since we have both categorical and numerical features, we perform this examination separately.

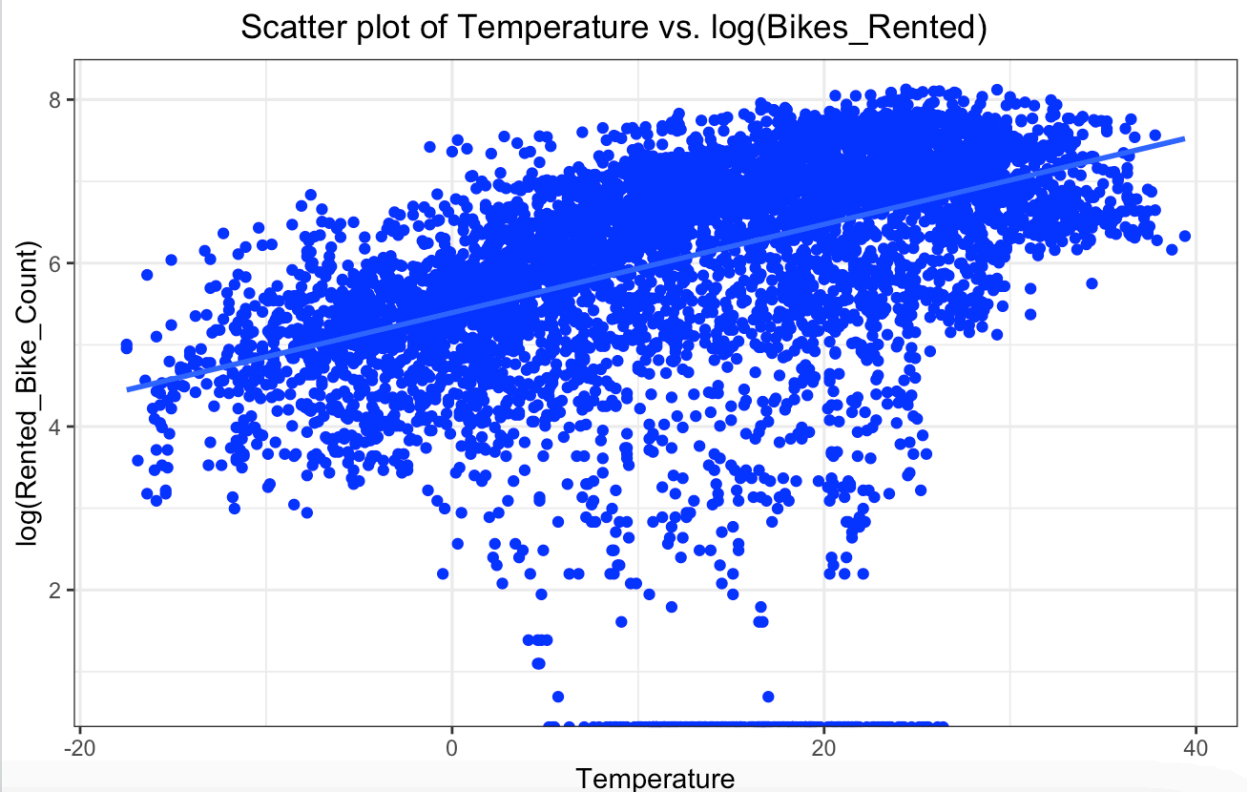
To identify highly correlated features, we first create a correlation matrix, which may be difficult to read due to its size. To make it easier to interpret, we use the caret package's findCorrelation() function. We set the cutoff to 0.75, and the function identifies the correlations above this threshold and suggests the variables to be removed. In this case, only dewpointemp is suggested to be removed. We keep this in mind and remove all redundant features together in our further analysis.

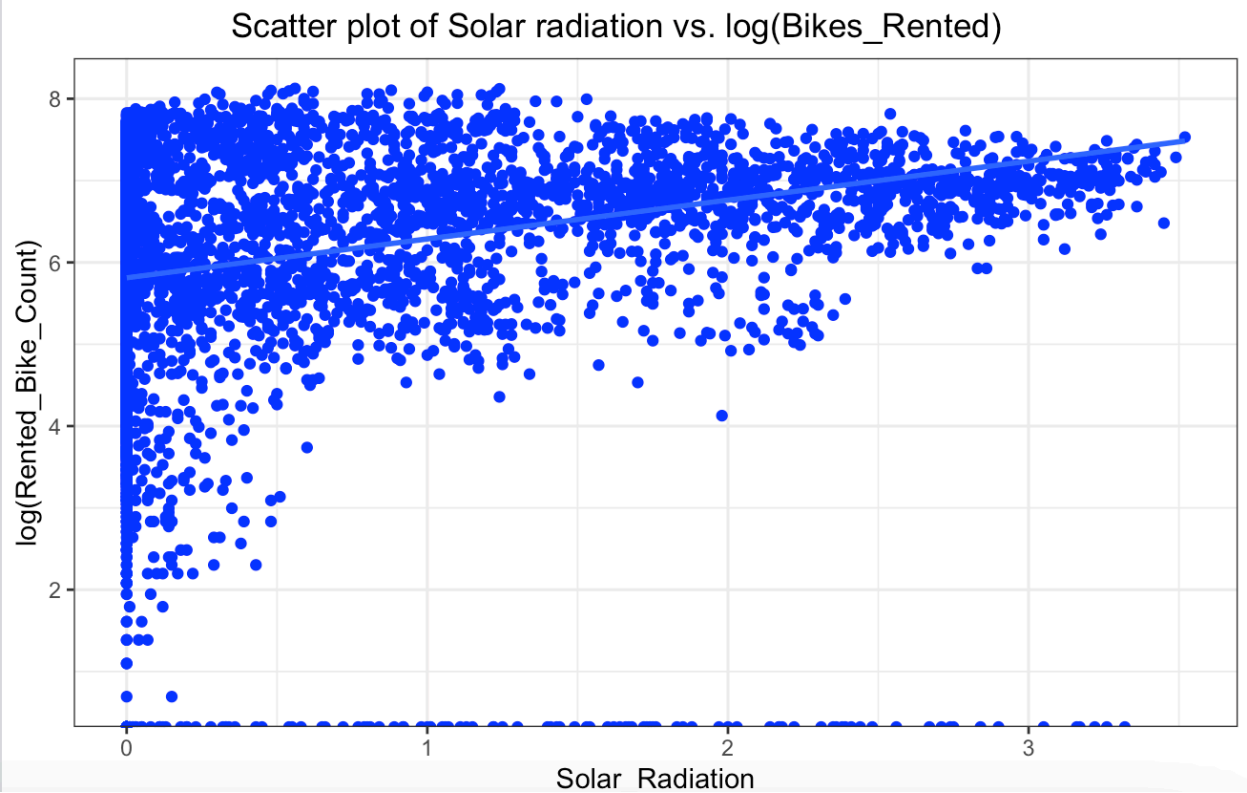
```
> # remove columns that have less correlation with target variable
> findCorrelation(bikes_correlation,
+                 cutoff = 0.75,
+                 names = TRUE) -> bikes_crr
> bikes_crr
[1] "Dew_point_temperature"
> # sort the correlations in ascending order with respect to target variable
> correlation_order <-
+   # we take correlations with number of bikes rented
+   bikes_correlation[,"Rented_Bike_Count"] %>%
+   # sort them in the decreasing order
+   sort(decreasing = TRUE) %>%
+   # end extract just variables' names
+   names()
> correlation_order
[1] "Rented_Bike_Count"     "Temperature"           "Dew_point_temperature"
[4] "Solar_Radiation"       "Visibility"            "Wind_speed"
[7] "Rainfall"              "Snowfall"              "Humidity"
> # plot some correlations
> #install.packages("ggplot2")
> #install.packages("corrplot")
> corrplot.mixed(bikes_correlation[correlation_order, correlation_order],
+                upper = "square", lower = "number",
+                tl.col="black", tl.pos = "lt")
```

We will sort the predictors based on their correlation with the outcome variable in a descending order, and then use this order to arrange the rows and columns of the correlation matrix plot. This will help us visualize the relationship between each predictor and the target variable (bikes rented).



The correlation matrix shows a strong correlation between dew point and temperature. To address this, we will consider both variables together in our final selection. Moving on, we can now examine the relationship between the top 3 predictors and the outcome variable (i.e., bikes rented) using scatter plots. The plots clearly demonstrate a positive relationship between these predictors and the outcome variable.


Scatter plot of Temperature vs. log(Bikes_Rented)
log(Rented_Bike_Count)
8
6
4
2
-20
0
20
40
Temperature


Scatter plot of Solar radiation vs. log(Bikes_Rented)
log(Rented_Bike_Count)
8
6
4
2
0
1
2
3
Solar Radiation

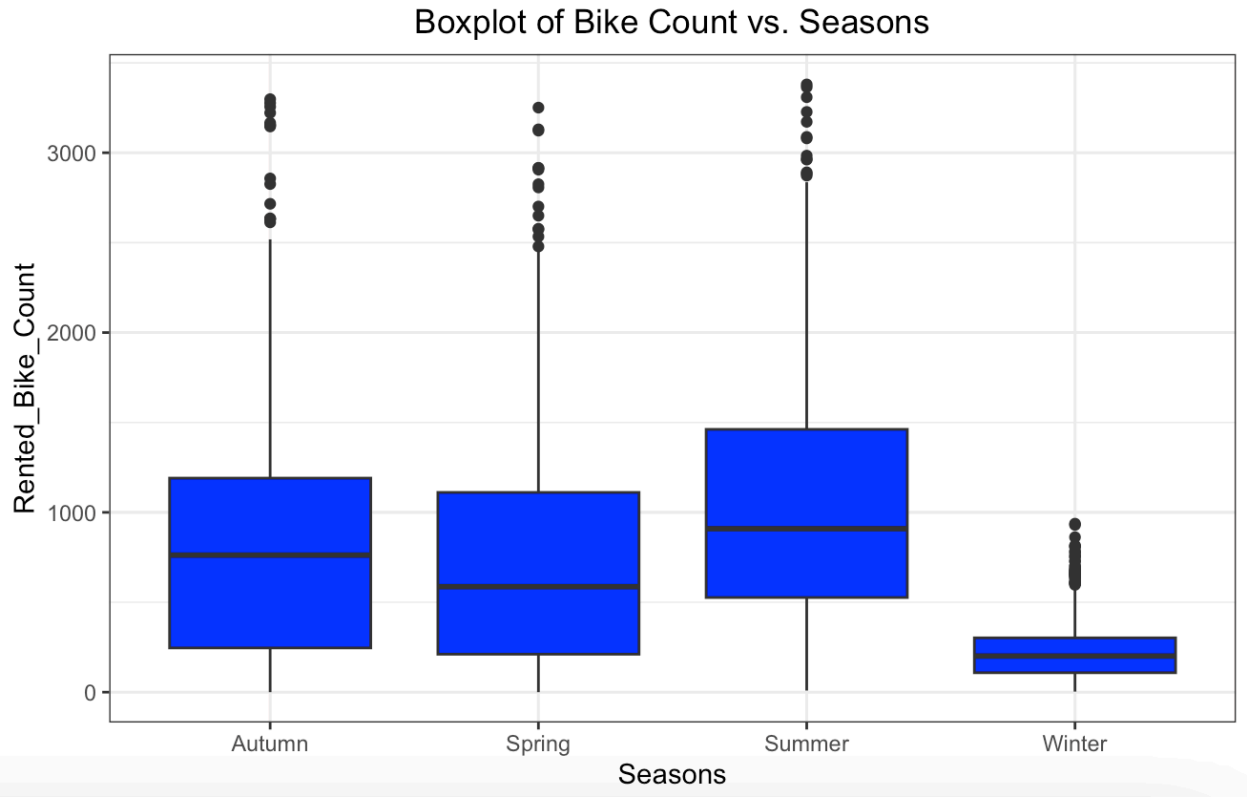
Correlation of Categorical Variables vs Target Variable:

```
> # correlation of categorical variables with respect to target variable
> bikes_F_anova <- function(bikes_categorical){
+   anova_ <- aov(train_data$Rented_Bike_Count ~
+                   train_data[[bikes_categorical]])
+
+   return(summary(anova_)[[1]][1, 5])
+ }
> sapply(bikes_categorical,
+        bikes_F_anova) %>%
+   # in addition lets sort them
+   # in the decreasing order of F
+   #  and store as an object
+   sort(decreasing = TRUE) -> bikes_anova_all_categorical
> bikes_anova_all_categorical
        Holiday Functioning_Day         Seasons            Hour
   1.652443e-06    1.971240e-58   6.343395e-311   4.051338e-322
> # Boxplot of Bike Count vs. Seasons
> ggplot(train_data,
+        aes(x = Seasons ,
+            y = Rented_Bike_Count)) +
+   geom_boxplot(fill = "blue") +
+   theme_bw()+
+   labs(title = "Boxplot of Bike Count vs. Seasons")+
+   theme(plot.title = element_text(hjust = 0.5))
```

Usage of ANOVA test:

- The ANOVA test is used in the above function because the aim is to test whether there is a significant difference in the mean 'Rented_Bike_Count' between different categorical variables.
- In this case, the 'bikes_categorical' variable holds different categorical variables, which are holiday, functioning day, seasons, and hour categories that may affect the number of rented bikes.
- The ANOVA test can be used to determine whether there is a significant difference in the mean 'Rented_Bike_Count' across different categorical variables, which would suggest that the variable is a significant predictor of bike rentals.

- Therefore, the ANOVA test is an appropriate statistical tool to use in this scenario to determine whether the categorical variable specified in the input argument of the bikes_F_anova() function is significantly associated with the 'Rented_Bike_Count' variable in the 'train_data' data frame.



To assess the impact of categorical predictors on the outcome variable, which is quantitative, we can use analysis of variance (ANOVA) to test the hypothesis that the mean number of rented bikes does not differ for different levels of the categorical predictor, such as hours or seasons. We applied a function to extract the p-value for all categorical predictors and sorted them in descending order based on p-value. All p-values were found to be less than the significance level of 0.05, indicating that we reject the null hypothesis and accept the alternative hypothesis, i.e., all categorical predictors have an impact on the outcome variable, which is the number of rented bikes.

We also plotted the box plot for one of the categorical predictors, such as seasons, to observe the relationship with the outcome variable. The box plot shows that the number of rented bikes varies with different seasons, with the summer season showing a higher number of rented bikes.

Additionally, we observed some strong correlation among the predictors, such as dew point and temperature, which we will address in the final selection of variables.

Identifying Linear Combinations:

Collinearity check:

- When two or more variables in a dataset are highly correlated with each other, they may provide redundant information, leading to overfitting and unstable estimates in statistical models. Collinearity can also make it difficult to interpret the relative importance of individual predictors in a model. Therefore, identifying and addressing collinearity is important to improve the accuracy and robustness of predictive models.

- We utilized the function findLinearCombos(data) to identify linear relationships in the numeric variables of our dataset. This function decomposes the data matrix and identifies groups of variables that can be expressed as linear combinations of other variables. Such variables are considered redundant as they do not add any information to the analysis.

```
> # look for collinearity between columns
> ( findLinearCombos(train_data[, bikes_numerical] ) ->
+     houses_linearCombos )
$linearCombos
list()

$remove
NULL
```

The function returned a null result, indicating that we do not have any linear combinations in our training data.

Identifying Features with Zero or Near Zero Variance:

Zero variance check:

- When a variable has zero variance or very low variance, it means that it does not vary much across the dataset and hence does not contain much information that can help in modeling. Such variables are generally not useful in predictive modeling and can even cause issues such as division by zero in certain algorithms. Identifying and removing zero-variance variables can help to simplify the model and improve its efficiency.

- One way to achieve this is to use the nearZeroVar() function from the caret package. This function can help identify and remove such variables from the dataset.

```
> # look for columns with zero variance
> nearZeroVar(train_data,
+             saveMetrics = TRUE) -> bikes_nzv_stats
> bikes_nzv_stats %>%
+   # we add rownames of the frame
+   # (with names of variables)
+   # as a new column in the data
+   tibble::rownames_to_column("variable") %>%
+   # and sort it in the descreasing order
+   arrange(-zeroVar, -nzv, -freqRatio)
                variable  freqRatio percentUnique zeroVar   nzv
1              Snowfall 181.937500    0.73373553   FALSE  TRUE
2              Rainfall  73.871795    0.79895647   FALSE  TRUE
3       Solar_Radiation  31.715789    5.57639002   FALSE  TRUE
4       Functioning_Day  28.771845    0.03261047   FALSE  TRUE
5            Visibility  65.125000   26.82210990   FALSE FALSE
6               Holiday  18.720257    0.03261047   FALSE FALSE
7     Rented_Bike_Count  13.733333   31.69737486   FALSE FALSE
8                  Hour   1.418283    0.04891570   FALSE FALSE
9  Dew_point_temperature   1.375000    8.85374205   FALSE FALSE
10          Temperature   1.068966    8.70699495   FALSE FALSE
11             Humidity   1.058824    1.45116582   FALSE FALSE
12           Wind_speed   1.036101    1.01092451   FALSE FALSE
13              Seasons   1.017453    0.06522094   FALSE FALSE
```

- Based on the output of the function nearZeroVar(data, saveMetrics = FALSE) from the caret package, it appears that there are 5 variables in our data with near zero variance.
- The five variables are “Solar Radiation”,”Rainfall”,”Snowfall”,”Functioning Day”, “Holiday”.

Final Selection of Features:

```
> selected_variables <- names(seoul_bike_data)
> selected_variables <- c(correlation_order,
+                         names(bikes_anova_all_categorical)
+ )
> selected_variables  <-
+   selected_variables [!selected_variables %in%
+                         bikes_crr]
> #lets now delete the near zero variance variables
> (bikes_variables_nzv <- nearZeroVar(train_data,
+                                     names = TRUE))
[1] "Solar_Radiation" "Rainfall"        "Snowfall"        "Functioning_Day"
> selected_variables  <-
+   selected_variables [!selected_variables  %in%
+                         bikes_variables_nzv]
> selected_variables
[1] "Rented_Bike_Count" "Temperature"       "Visibility"        "Wind_speed"
[5] "Humidity"          "Holiday"           "Seasons"           "Hour"
> #apply the backward elimination method and find out the redundant features and remo
ve them
> bikes_lm1 <- lm(Rented_Bike_Count ~ .,
+                 data = train_data %>%
+                   dplyr::select(all_of(selected_variables)))
```

```
> vifs <- ols_vif_tol(bikes_lm1)
> vifs[which(vifs$Variables %in% bikes_numerical),]
    Variables Tolerance      VIF
1 Temperature 0.2328268 4.295038
2  Visibility 0.5933937 1.685222
3  Wind_speed 0.8209909 1.218040
4    Humidity 0.4951319 2.019664
```

Variance Inflation Factor:

Variance inflation factor (VIF) is a measure of the degree of multicollinearity between predictor variables in a multiple regression model. VIF is calculated by regressing each predictor variable against all other predictor variables in the model and then computing the ratio of the variance of the coefficient estimates to the variance of a hypothetical coefficient estimate if the predictor variable were not correlated with the other predictors.

In other words, VIF measures how much the variance of the estimated regression coefficient for a particular predictor variable is inflated due to multicollinearity with the other predictor variables in the model. A high VIF value (greater than 5 or 10, depending on the context) indicates that the predictor variable is highly correlated with the other predictor variables and may cause problems in the interpretation of the regression coefficients or in making accurate predictions. Therefore, variables with high VIF values are often removed from the model to reduce multicollinearity and improve model performance.

The output above displays the variance inflation factor (VIF) value for all predictors. The VIF values are relatively low, indicating that there is no serious issue of multicollinearity. We will now implement the backward elimination method to identify redundant features and eliminate them.

Code Explanation:

- This above R code performs multiple linear regression using the lm() function to predict the number of rented bikes based on several independent variables. The backward elimination method is applied to identify and remove redundant variables from the model.
- Firstly, the lm() function is used to fit a linear model named bikes_lm1 that regresses the Rented_Bike_Count variable on all the variables in the train_data dataset selected by the selected_variables vector.
- Then, the olsrr package is loaded to calculate the Variance Inflation Factor (VIF) of the numerical variables in the model. High VIF values indicate the presence of collinearity between variables.

- Next, the ols_vif_tol() function is used to calculate the VIF values and a subset of VIFs for numerical variables is printed using which() and %in% functions.
- Then, the ols_step_backward_p() function is used to perform backward elimination using a significance level of 0.05 (prem = 0.05) and the model is stored in bikes_lm1_backward_p.
- The summary() function is used to display the summary statistics of the final model after the backward elimination procedure.
- The removed component of the bikes_lm1_backward_p object displays the variables that were removed during the backward elimination process.
- Finally, the selected_final_variables vector is created by removing the variables "Visibility" and "Wind_speed" from the original selected_variables vector as these variables were identified as redundant by the backward elimination method.

```
> bikes_lm1_backward_p <- ols_step_backward_p(bikes_lm1,
+                                             prem = 0.05,
+                                             progress = F)
> summary(bikes_lm1_backward_p$model)

Call:
lm(formula = paste(response, "~", paste(preds, collapse = " + ")),
    data = l)

Residuals:
     Min       1Q   Median       3Q      Max
-1640.51  -258.92    -7.21   237.04  1964.67

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)        720.8459    36.6785  19.653  < 2e-16 ***
Temperature         25.0533     0.9708  25.807  < 2e-16 ***
Humidity            -8.7845     0.3031 -28.985  < 2e-16 ***
HolidayNo Holiday  119.7421    25.7199   4.656 3.30e-06 ***
SeasonsSpring      -64.6387    15.7154  -4.113 3.96e-05 ***
SeasonsSummer      -52.0940    20.0177  -2.602  0.00928 **
SeasonsWinter     -247.2216    23.0524 -10.724  < 2e-16 ***
HourHigh           437.1867    13.7457  31.805  < 2e-16 ***
Hourlow           -120.1296    16.1976  -7.416 1.37e-13 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 437.4 on 6124 degrees of freedom
Multiple R-squared:  0.5401,    Adjusted R-squared:  0.5395
F-statistic: 898.8 on 8 and 6124 DF,  p-value: < 2.2e-16

> #print the removed features
> bikes_lm1_backward_p$removed
[1] "Wind_speed" "Visibility"
> #list of selected_final_variables to be used for regression purpose
> selected_final_variables <- selected_variables[-which(selected_variables %in%
+                                                       c("Visibility","Wind_spee
d"))]
> #selected_variables
> selected_final_variables
[1] "Rented_Bike_Count" "Temperature"       "Humidity"          "Holiday"
[5] "Seasons"           "Hour"
```

After applying the backward elimination method, we have identified a list of features that are suggested to be removed. The remaining variables are then selected and stored in a list called ‘selected_final_variables’, which will be used for regression analysis. The regression estimates

indicate that all the 'selected_final_variables' have a significant relationship with the outcome variable. Our objective is to generate predictions for both training and testing data, so we will proceed to do so and estimate the root mean square error (RMSE) for both sets of data.

```
> #generate predictions and estimates RMSE on both training and testing data
> bikes_lm_benchmark$fitted.values <- exp(bikes_lm_benchmark$fitted.values)-1
> # function to calculate Regression Metrics
> regression_metrics <- function(real, predicted) {
+   # Compute residuals
+   residuals <- predicted - real
+
+   # Compute metrics
+   mse <- mean(residuals^2)
+   rmse <- sqrt(mse)
+   mae <- mean(abs(residuals))
+   msle <- mean((log(predicted + 1) - log(real + 1))^2)
+   r2 <- summary(lm(real ~ predicted))$r.squared
+
+   # Return metrics as a data frame
+   metrics_df <- data.frame(MSE = mse, RMSE = rmse, MAE = mae, MSLE = msle, R2 = r2)
+   return(metrics_df)
+ }
> # apply regression metrics function on train data
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bikes_lm_benchmark,
+                                        train_data[, selected_final_variables]))
       MSE     RMSE      MAE     MSLE        R2
1 901991.1 949.7321 698.5681 18.10417 0.4456368
> # apply regression metrics function on test data
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bikes_lm_benchmark,
+                                        test_data[, selected_final_variables]))
       MSE     RMSE      MAE     MSLE        R2
1 907202.8 952.4719 700.5987 18.17483 0.4333612
```

The RMSE for training data is 949 and for testing data it is 952. This indicates that linear regression performs well on both training and testing data. However, to control or minimize overfitting, we will try different machine learning algorithms and evaluate their results.

Regression using Decision Trees:

The process of building a simple regression tree is like that of a simple classification tree. However, it is not typically used on its own and is often utilized in combination with bagged, random forest, and gradient boosting techniques. Initially, a full tree is constructed, followed by k-fold cross-validation to determine the optimal cost complexity (cp) value. The primary distinction between simple classification and regression trees is that the rpart() parameter method is set to "anova" to create a regression tree.

```
> model.formula <- Rented_Bike_Count ~.
> # Regression Trees
> set.seed(1234)
> library(rpart)
> bikes.tree <- rpart(model.formula,
+                     data = train_data,
+                     method = "anova")
> summary(bikes.tree)
```

```
> #Lets generate the predictions and RMSE on both training and test data.
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree,
+                                        train_data))
      MSE     RMSE      MAE     MSLE        R2
1 119784.5 346.0989 244.6722 1.141414 0.7115454
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree,
+                                        test_data))
      MSE     RMSE     MAE     MSLE        R2
1 137302.3 370.5432 256.814 1.173874 0.6723069
```

The RMSE of the decision tree model is better than the linear regression benchmark model, but it appears that there is some degree of overfitting as the model's performance is not equally good on the testing data. To address this, we can apply cross-validation and tune the grid to find the optimal hyperparameters for the model.

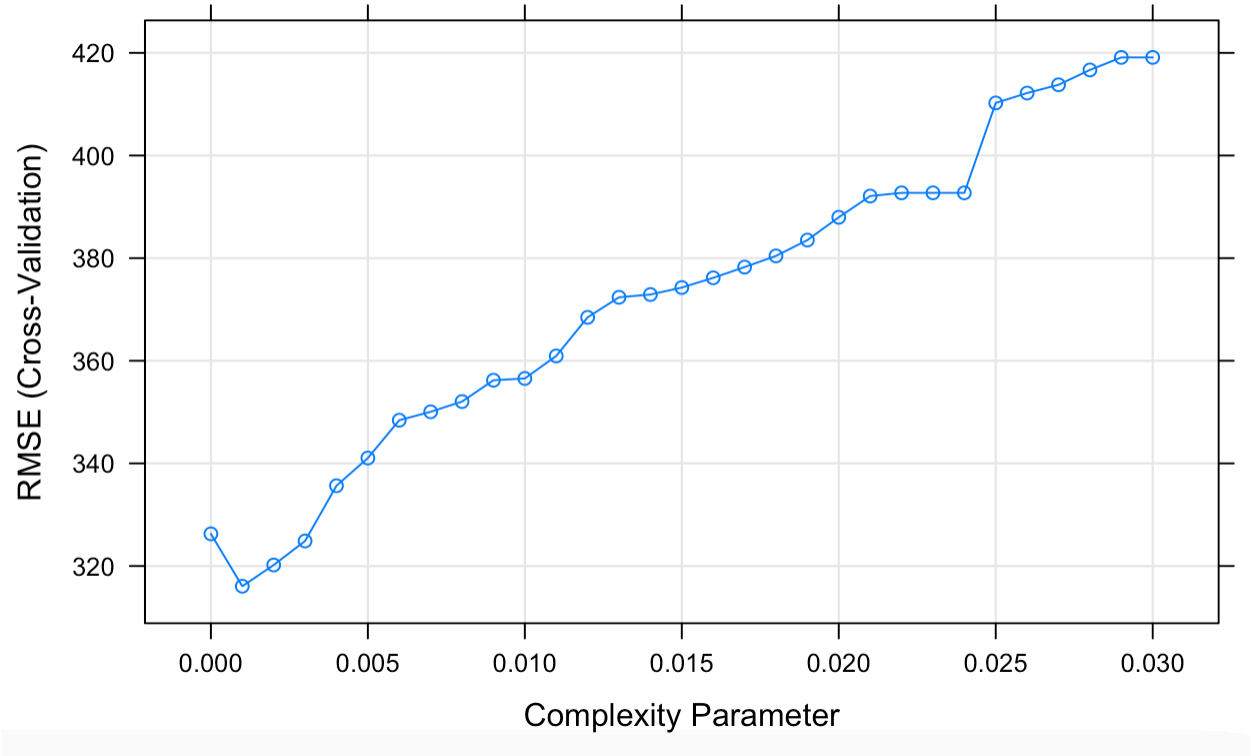
```
> #Lets try to apply cross validation and tune grid.
> tc <- trainControl(method = "cv", number = 10)
> cp.grid <- expand.grid(cp = seq(0, 0.03, 0.001))
> #39
> set.seed(123456789)
> bikes.tree_tuned <- train(model.formula,
+                           data = train_data,
+                           method = "rpart",
+                           trControl = tc,
+                           tuneGrid = cp.grid)
> #Lets plot and see how RMSE value responds to change in complexity parameter
> plot(bikes.tree_tuned)
> #calculate RMSE values on for training and testing data by using tuned model
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_tuned,
+                                        train_data))
       MSE     RMSE      MAE      MSLE        R2
1 86751.65 294.5363 203.1937 0.6394772 0.7910922
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_tuned,
+                                        test_data))
       MSE     RMSE      MAE      MSLE       R2
1 113354.2 336.6812 222.6324 0.5631734 0.729924
```

The results indicate a significant reduction in the RMSE value for the bike.tuned model on the training data, with a value of 294 compared to the linear regression and full decision tree models. However, this model appears to be the most overfitted as the RMSE value on the testing data is 336, which is the highest compared to the other models.

Code Explanation:

- The above code performs cross-validation and grid search to tune the hyperparameter of a decision tree model. Specifically, it uses the train() function from the caret package in R to train a decision tree using the rpart algorithm.
- trainControl() is used to specify the type of resampling method for cross-validation and the number of folds to use. Here, the method is set to "cv" for cross-validation, and the number of folds is set to 10.

- expand.grid() is used to create a grid of possible values for the hyperparameter cp (complexity parameter) to search over. The seq() function generates a sequence of values from 0 to 0.03, incrementing by 0.001 at each step.
- set.seed() is used to set a random seed for reproducibility.
- The train() function is then called, passing in the formula for the model, the training data, the resampling method, and number of folds specified in trainControl(), and the grid of hyperparameters specified in tuneGrid. The resulting model is stored in the bikes.tree_tuned object.
- Finally, plot() is used to create a plot of the results of the grid search. The plot shows the values of the hyperparameter cp on the x-axis and the root mean squared error (RMSE) on the y-axis. The plot helps to visualize how the performance of the model changes as the complexity parameter is varied, and can be used to identify the optimal value for cp.



The graph presented above indicates that the lowest RMSE is observed for cp=0.001, and for all other values of cp, the RMSE increases. Our next step is to generate predictions and evaluate RMSE values for both training and testing data using the tuned model.

Bagging:

- Bagging, short for bootstrap aggregation, is an ensemble machine learning technique used to improve the stability and accuracy of a model.
- The basic idea behind bagging is to create multiple subsets of the training data by randomly sampling with replacement from the original dataset. Each of these subsets is then used to train a different model, and the predictions from these models are combined to form the final prediction.
- By training multiple models on different subsets of the training data, bagging helps to reduce overfitting and improve the generalization ability of the model. It is particularly effective when the underlying model is unstable, meaning that small changes in the training data can result in large changes in the predictions.
- To use the bagging method, we will specify method = “treebag” and set tuneLength = 5 without any further tuning using tuneGrid. This is because Caret does not have any hyperparameters to tune for this model.

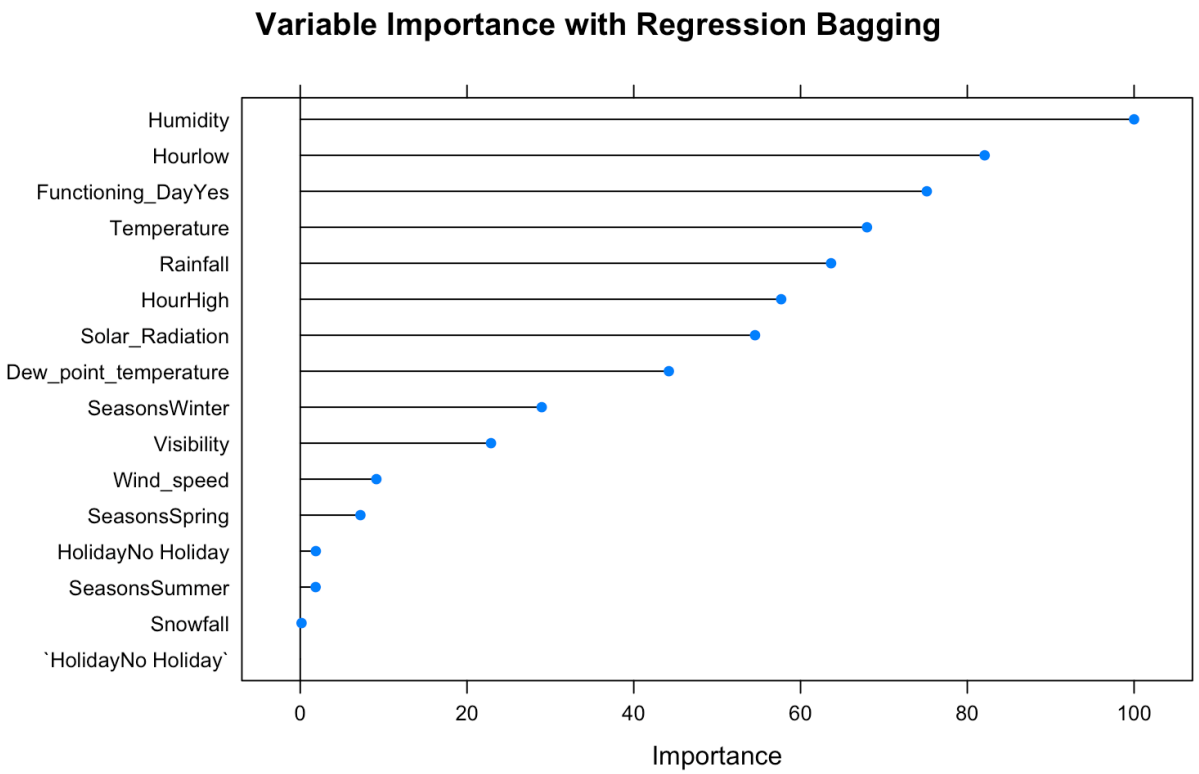
```
> # This time we will use the bagging method by specifying method = “treebag”
> bikes.tree_bagged <- train(model.formula,
+                            data = train_data,
+                            method = "treebag",
+                            trControl = tc)
> bikes.tree_bagged
Bagged CART

6133 samples
  12 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 5518, 5520, 5520, 5519, 5518, 5521, ...
Resampling results:

  RMSE     Rsquared   MAE
  340.162  0.7226904  239.84

> plot(varImp(bikes.tree_bagged), main="Variable Importance with Regression Bagging")
> # Lets now generate RMSE value for bagged model
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_bagged,
+                                        train_data))
       MSE     RMSE      MAE     MSLE        R2
1 112647.2 335.6296 237.3765 1.190215 0.7299042
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_bagged,
+                                        test_data))
       MSE     RMSE     MAE     MSLE        R2
1 127502.7 357.0751 247.376 1.169345 0.6961245
```



The output suggests a minor difference, indicating a slight issue of overfitting.

Random Forest

Apply Random Forest with Cross Validation

- Random Forest is an ensemble machine learning algorithm that is widely used for classification and regression tasks. It works by building many decision trees and then combining their outputs to make a final prediction.

- Cross-validation is a technique used to evaluate the performance of a machine learning model. It involves dividing the dataset into several folds, training the model on one-fold, and evaluating its performance on the other folds.

- This process is repeated several times, and the results are averaged to get a more accurate estimate of the model's performance.

- By using cross-validation, we can better estimate the performance of our Random Forest model on unseen data and reduce the risk of overfitting to the training data.

```
> #lets now apply random forest with cross validation
> set.seed(71)
> bikes.tree_rf <- train(model.formula,
+                        data = train_data,
+                        method = "ranger",
+                        trControl = tc)
> bikes.tree_rf
Random Forest

6133 samples
  12 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 5519, 5520, 5521, 5519, 5519, 5519, ...
Resampling results across tuning parameters:

  mtry  splitrule   RMSE      Rsquared   MAE
   2    variance    316.5754  0.7818402  221.3855
   2    extratrees  347.7862  0.7398438  245.2312
   8    variance    276.5309  0.8163704  180.4433
   8    extratrees  274.6953  0.8192256  179.9764
  15    variance    280.3086  0.8112899  182.0200
  15    extratrees  272.7174  0.8215741  178.0506

Tuning parameter 'min.node.size' was held constant at a value of 5
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were mtry = 15, splitrule = extratrees
 and min.node.size = 5.
> plot(bikes.tree_rf)
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_rf,
+                                        train_data))
       MSE     RMSE      MAE      MSLE        R2
1 18864.98 137.3499 89.50433 0.1447809 0.9571529
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bikes.tree_rf,
+                                        test_data))
       MSE   RMSE      MAE      MSLE       R2
1 79388.69 281.76 183.3847 0.3324507 0.809947
```

```
> #Lets tuned this model and try out different combination of parameters
> bkes_rf_tuned_1 <- randomForest(model.formula,
+                                 data = train_data,
+                                 ntree = 100,
+                                 mtry = 8,
+                                 # minimum number of obs in the terminal nodes
+                                 nodesize = 100,
+                                 # we also generate predictors importance measures,
+                                 importance = TRUE)
> print(bkes_rf_tuned_1)

Call:
 randomForest(formula = model.formula, data = train_data, ntree = 100,      mtry = 8,
nodesize = 100, importance = TRUE)
               Type of random forest: regression
                     Number of trees: 100
No. of variables tried at each split: 8

          Mean of squared residuals: 82560.18
                    % Var explained: 80.12
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_1,
+                                        train_data))
       MSE    RMSE      MAE      MSLE        R2
1 66702.88 258.269 172.1418 0.6469716 0.8406842
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_1,
+                                        test_data))
       MSE     RMSE      MAE      MSLE        R2
1 89604.84 299.3407 196.9153 0.6830023 0.7859636
```

Code Explanation:

Let's improve the performance of the random forest model by tuning its parameters and testing different combinations.

The code above is tuning a Random Forest model using different combinations of hyperparameters. Here is a brief explanation of the code:

This is a comment indicating that the following code will tune the model.

This creates the first tuned Random Forest model with the following parameters:

model.formula: A formula specifying the response variable and the predictor variables to be used in the model.

data: The training data.

ntree: The number of trees to grow in the forest.

mtry: The number of variables randomly sampled as candidates at each split.

nodesize: The minimum number of observations in the terminal nodes.

importance: A logical value indicating whether to compute measures of variable importance.

What is 'mtry' parameter in RandomForest function?

- In a Random Forest model, the trees are grown by recursively partitioning the feature space into smaller and smaller subspaces. At each split, the algorithm must select a subset of the available features (i.e., variables) to use as candidates for the split.
- The 'mtry' parameter controls how many variables are included in this subset.
- A common rule of thumb is to set 'mtry' to the square root of the total number of features in the dataset.
- For example, if there are 16 features, 'mtry' would be set to 4. This ensures that a wide range of variables are considered at each split, but not so many that the trees become overly correlated with each other
- In general, setting 'mtry' too low can lead to underfitting, as important variables may be missed. Setting 'mtry' too high can lead to overfitting, as noise variables may be included. Therefore, finding the optimal value for 'mtry' is an important tuning parameter for Random Forest models.

```
> #
> bkes_rf_tuned_2<- randomForest(model.formula,
+                                data = train_data,
+                                ntree = 100,
+                                mtry = 6,
+                                # minimum number of obs in the terminal nodes
+                                nodesize = 100,
+                                # we also generate predictors importance measures,
+                                importance = TRUE)
> print(bkes_rf_tuned_2)

Call:
 randomForest(formula = model.formula, data = train_data, ntree = 100,      mtry = 6,
nodesize = 100, importance = TRUE)
               Type of random forest: regression
                     Number of trees: 100
No. of variables tried at each split: 6

          Mean of squared residuals: 83132.67
                    % Var explained: 79.98
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_2,
+                                        train_data))
       MSE     RMSE      MAE     MSLE        R2
1 67622.82 260.0439 174.2315 0.866244 0.8395111
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_2,
+                                        test_data))
       MSE     RMSE      MAE     MSLE        R2
1 89339.51 298.8971 198.1263 0.938371 0.7873236
```

- This time we have tuned Random Forest model with a different ‘mtry’ parameter.

```
> parameters_rf <- expand.grid(mtry = 2:12)
> #
> bkes_rf_tuned_3 <-
+   train(model.formula,
+         data = train_data,
+         method = "rf",
+         ntree = 100,
+         nodesize = 100,
+         tuneGrid = parameters_rf,
+         trControl = tc)
> regression_metrics(real = train_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_3,
+                                        train_data))
       MSE    RMSE      MAE      MSLE        R2
1 66213.58 257.32 171.6654 0.5405032 0.8417459
> regression_metrics(real = test_data$Rented_Bike_Count,
+                    predicted = predict(bkes_rf_tuned_3,
+                                        test_data))
       MSE     RMSE      MAE      MSLE        R2
1 90135.35 300.2255 197.2142 0.6095759 0.7844845
> plot(bkes_rf_tuned_3)
```

Code Explanation:

This creates the third tuned Random Forest model using the train() function from the caret package, which allows for more sophisticated tuning of hyperparameters. The following parameters are used:

model.formula: A formula specifying the response variable and the predictor variables to be used in the model.

data: The training data.

method: The machine learning method to be used, in this case "rf" for Random Forest.

ntree: The number of trees to grow in the forest.

nodesize: The minimum number of observations in the terminal nodes.

tuneGrid: A set of hyperparameters to be tuned.

trControl: A control object specifying the resampling method to be used for tuning.

Overall, the code is trying to find the optimal combination of hyperparameters for the Random Forest model to achieve the best predictive performance on the training data.

Summary and Conclusions:

We will now compare the results of various estimated models, including the linear regression benchmark (bikes_lm_benchmark), decision tree (bikes.tree), tuned decision tree (bikes.tree_tunned), bagged model (bikes.tree_bagged), and random forest. To do this, we will compute predicted values for each model and summarize the error measures.

```
> # Summary and Conclusions
> bikes_model_list <- list(bikes_lm_benchmark =bikes_lm_benchmark ,
+                          bikes.tree=bikes.tree,
+                          bikes.tree_tuned= bikes.tree_tuned,
+                          bikes.tree_bagged=bikes.tree_bagged,
+                          bikes.tree_rf=bikes.tree_rf,
+                          bkes_rf_tuned_1=bkes_rf_tuned_1,
+                          bkes_rf_tuned_2=bkes_rf_tuned_2,
+                          bkes_rf_tuned_3=bkes_rf_tuned_3)
> bikes_fitted <- bikes_model_list %>%
+   sapply(function(x) predict(x, newdata=train_data)) %>%
+   data.frame()
> sapply(bikes_fitted,
+        function(x) regression_metrics(real = train_data$Rented_Bike_Count, predicte
d = x
+        )) %>%  t()
                    MSE      RMSE     MAE      MSLE      R2
bikes_lm_benchmark 901991.1 949.7321 698.5681 18.10417  0.4456368
bikes.tree         119784.5 346.0989 244.6722 1.141414  0.7115454
bikes.tree_tuned   86751.65 294.5363 203.1937 0.6394772 0.7910922
bikes.tree_bagged  112647.2 335.6296 237.3765 1.190215  0.7299042
bikes.tree_rf      18864.98 137.3499 89.50433 0.1447809 0.9571529
bkes_rf_tuned_1    66702.88 258.269  172.1418 0.6469716 0.8406842
bkes_rf_tuned_2    67622.82 260.0439 174.2315 0.866244  0.8395111
bkes_rf_tuned_3    66213.58 257.32   171.6654 0.5405032 0.8417459
> bikes_fitted <- bikes_model_list %>%
+   sapply(function(x) predict(x, newdata=test_data)) %>%
+   data.frame()
> sapply(bikes_fitted,
+        function(x) regression_metrics(real = test_data$Rented_Bike_Count, predicted
= x
+        )) %>%  t()
                    MSE      RMSE     MAE      MSLE      R2
bikes_lm_benchmark 907202.8 952.4719 700.5987 18.17483  0.4333612
bikes.tree         137302.3 370.5432 256.814  1.173874  0.6723069
bikes.tree_tuned   113354.2 336.6812 222.6324 0.5631734 0.729924
bikes.tree_bagged  127502.7 357.0751 247.376  1.169345  0.6961245
bikes.tree_rf      79388.69 281.76   183.3847 0.3324507 0.809947
bkes_rf_tuned_1    89604.84 299.3407 196.9153 0.6830023 0.7859636
bkes_rf_tuned_2    89339.51 298.8971 198.1263 0.938371  0.7873236
bkes_rf_tuned_3    90135.35 300.2255 197.2142 0.6095759 0.7844845
```

After applying different models, the results indicate that the random forest model (bikes.tree_rf) has the lowest RMSE on the training data.

However, it performs the worst on the testing data compared to other model results, indicating significant overfitting.

While all the models show some degree of overfitting, we will select the random forest model (bikes_rf_tuned_1) based on its lower RMSE for both training (258) and testing (299) data, and relatively lesser overfitting. Therefore, we can conclude that this model could perform equally well on unseen data.

Bibliography:

R Libraries Used:

dplyr: A package for data manipulation and transformation. It provides a set of functions for filtering, arranging, summarizing, and joining datasets.

caret: A package for machine learning and predictive modelling. It provides a unified interface for training and evaluating models using various algorithms and techniques.

ggplot2: A package for data visualization. It provides a grammar of graphics framework for creating high-quality, customizable plots and charts.

corrplot: A package for visualizing correlation matrices. It provides a variety of plot types and customization options for exploring relationships between variables.

ranger: A package for random forest classification and regression. It provides a fast implementation of the random forest algorithm with advanced features such as variable importance measures and out-of-bag error estimates.

olsrr: A package for linear regression diagnostics and visualization. It provides a range of functions for assessing model assumptions, checking for multicollinearity, and visualizing regression results.

rpart: A package for decision tree classification and regression. It provides a simple and flexible implementation of the decision tree algorithm for predicting outcomes based on a set of predictor variables.

randomForest: A package for random forest classification and regression. It provides an implementation of the random forest algorithm for building ensembles of decision trees and making predictions based on their combined outputs.

Tools & Frameworks

R studio and Jupiter Notebook

References:

https://archive.ics.uci.edu/ml/datasets/Seoul+Bike+Sharing+Demand

https://www.tandfonline.com/doi/full/10.1080/22797254.2020.1725789

https://rpubs.com/asmi2990/869714

https://new.ubreakifix.com/?_ga=2.141521073.283448484.1677950172-885920165.1677950171&device=1354&postal_code=76201&repair=4619&service=CC