BIKE RENTING

Santosh Kamat

Contents

Chapter 1

Introduction

1.1. Problem Statement

We are provided with daily bike rental data spanning for 2 years. Our objective is to predict daily rental count based on seasonal and environmental settings

1.2. Data

Our task is to build regression models which will predict the bike rental count based considering the seasonal and environmental factors

Table 1.1 **Bike Rental Data** (Columns 1-8)

instant	dteday	season	yr	mnth	holiday	weekday	workingday
1	1/1/2011	1	0	1	0	6	0
2	1/2/2011	1	0	1	0	0	0
3	1/3/2011	1	0	1	0	1	1
4	1/4/2011	1	0	1	0	2	1
5	1/5/2011	1	0	1	0	3	1
6	1/6/2011	1	0	1	0	4	1

Table 1.2 **Bike Rental Data** (Columns 9-16)

weatersit	temp	atemp	hum	windspeed	casual	registered	cnt
2	0.344167	0.363625	0.805833	0.160446	331	654	985
2	0.363478	0.353739	0.696087	0.248539	131	670	801
1	0.196364	0.189405	0.437273	0.248309	120	1229	1349
1	0.2	0.212122	0.590435	0.160296	108	1454	1562
1	0.226957	0.229270	0.436957	0.1869	82	1518	1600
1	0.204348	0.233209	0.518261	0.0895652	88	1518	1606

The columns/variables present in the dataset are: instant, dteday, season, mnth, holiday, weekday, workingday, weathersit, temp, atemp, hum, windspeed, casual, registered, cnt

The details of the attributes in the dataset are as follows:

instant: record Index

dteday: Date

season: Season (1: Springer, 2: Summer, 3: fall, 4: winter)

yr: Year (0:2011, 1:2012)

mnth: (Month 1-12)

holiday: whether a day is a holiday or not

weekday: day of the week

workingday: If a day is neither weekend nor holiday is 1, otherwise 0

weathersit: (Extracted from Freemeteo)

1: Clear, Few Clouds, Partly Cloudy

2: Mist + Cloudy, Mist + broken clouds, Mist + few clouds, Mist

3: Light snow, Light rain + Thunderstorm + scattered clouds, Light rain + scattered clouds

4: Heavy rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog

temp: Normalized temperature in Celsius. The values are derived via

(t-t_min)/(t_max-t_min) where t_min = 8 and t_max = 39 (only in hourly scale)

atemp: Normalized feeling temperature in Celsius. The values are derived via

(t-t_min)/(t_max-t_min) where t_min=16 and t_max=50 (only in horly scale)

hum: Normalized humidity. The values are divided by 100 (max)

windspeed: Normalized humidity. The values are divided by 67 (max)

casual: count of casual users

registered: count of registered users

cnt: count of total rental bikes including both casual and registered

Chapter 2

Methodology

2.1. Pre-Processing

Pre-processing refers to the transformations applied to our data before feeding it to the machine learning algorithm. Data pre-processing is a technique that is used to convert raw data into clean data set. It involves Missing Value Analysis, Outlier Analysis, tackling Multicollinearity, Exploratory Data Analysis and Feature Selection.

2.1.1. **Missing Value Analysis**

Missing Value Analysis is done to ensure that there are no missing values in our data set. There are many ways to deal with the missing values. One way is to remove the rows or records containing the missing values or drop the column depending on the count of missing values. Other way is to impute the missing value using mean, median, KNN imputation.

Python code

In python we make use of **pandas** library to work with data frames. Also we require **os** library to set the current working directory.

```
import os
import pandas as pd
```

We read the csv file from the current working directory

```
# set the working directory
os.chdir("C:/Users/Pranav/Desktop/Bike Rental Project")
```

```
# Read data file
data =pd.read_csv('day.csv')
```

Now we can analyse how many missing values are there in the data set by doing as follows

```
In [7]: data.isnull().sum()
Out[7]: instant       0
        dteday        0
        season        0
        yr            0
        mnth          0
        holiday       0
        weekday       0
        workingday    0
        weathersit    0
        temp          0
        atemp         0
        hum           0
        windspeed     0
        casual        0
        registered    0
        cnt           0
        dtype: int64
```

We see that there are no missing values in the data set

R Code

First, we need to set the current working directory.

```
setwd("C:/Users/Pranav/Desktop/Bike Rental Project")
```

We read the csv file as follows using the **read.csv()** function

```
## Read the data from csv file
data = read.csv("day.csv", header = T)
```

Next we check the missing values in the data frame using **sapply()** function

```
> sapply(data,function(x) sum(is.na(x)))
   instant     dteday     season         yr       mnth    holiday    weekday workingday weathersit       temp      atemp        hum
         0          0          0          0          0          0          0          0          0          0          0          0
 windspeed     casual registered        cnt
         0          0          0          0
```

We see that there are no missing values in our data.

2.1.2. Removing Redundant Variables

In this step we remove the variables that will play no role in predictions. We also remove the variables which are redundant in nature. Redundant variables are responsible for increasing the computational time and hamper the efficiency of the model.

In our case, "**instant**" variable has no role in predictions and hence can be removed. Moreover "**casual**" and "**registered**" are the variables whose sum evaluates to dependent variable "**cnt**". If "**casual**" + "**registered**" = "**cnt**", it doesn't make any sense to apply machine learning algorithm to predict '**cnt'.** Hence we have to drop "**casual**" and "**registered**" variables. Furthermore "**dteday**" is a redundant variable as we already have information about year, month, holiday,etc. Thus "**dteday**" can be dropped.

Python code

We can drop the columns using **drop()** function in pandas.

```
data = data.drop(columns = ["instant","dteday","casual","registered"])
```

R code

By doing the following we can drop those columns in R

```
# remove dteday as it is redundant.We already have yr, mnth, holiday, weekday, workingday as attributes
# we also remove instant as it has no use in predictions
data$dteday = NULL
data$instant = NULL

# we also remove casual and registered as casual + registered = cnt.
# if such is the case applying machine learning algoruthms for predictions make no sense
data$registered = NULL
data$casual = NULL
```

2.1.3. Outlier Analysis

Outliers are extreme values that deviate from other observations in the data. The rows containing outliers can either be deleted or the outliers can be imputed with mean, median, KNN imputation, etc.

Boxplot is a standard way to detect outliers

Maximum = Q3 + IQR * 1.5

Minimum = Q1 – IQR * 1.5

Here Q3 is the third quartile (75^{th} percentile), Q1 is the first quartile (25^{th} percentile), IQR is the interquartile range (Q3-Q1)

The observations which are greater than **Maximum** or less than the **Minimum** are considered to be outliers

We have done **mean imputation** for outliers. The outliers have been replaced by the mean of the corresponding column.

Python code

To plot the boxplot, we have used **seaborn** library. First we need to import this library.

```
import seaborn as sn
```

Currently, we have 4 independent continuous variables with us. They are "**temp**","**atemp**","**hum**","**windspeed**"

We can plot the boxplot using **boxplot()** function in seaborn library. This can be done as follows.

```
numerical_vars = ['temp','atemp', 'hum', 'windspeed']

for col in numerical_vars:
    sn.boxplot(data[col])
    plt.show()
```

Following are the 4 boxplots generated

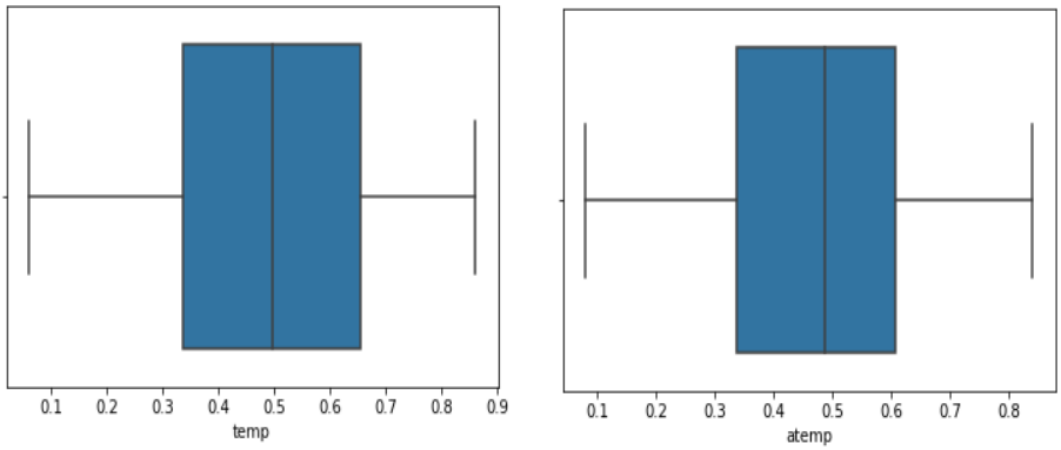

Fig 1: Boxplot of "**temp**" and "**atemp**" variables

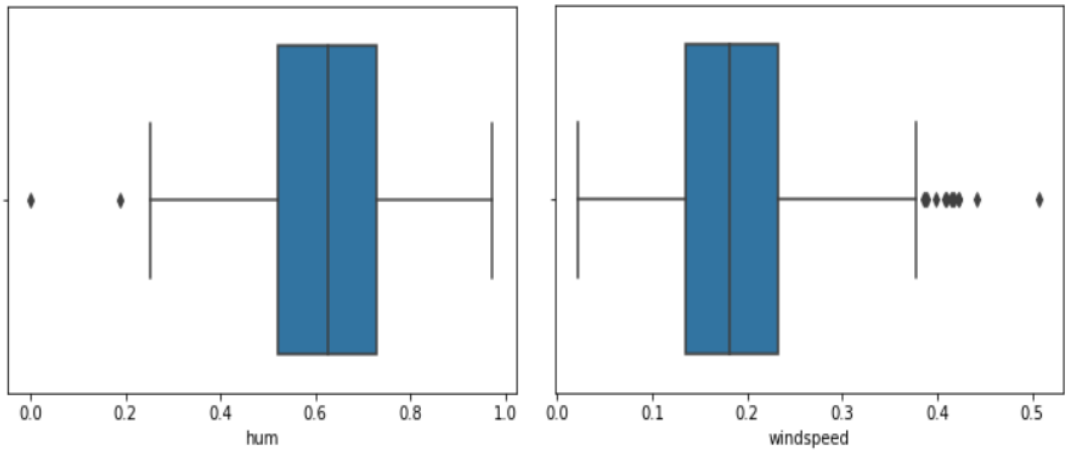

Fig 2: Boxplot of "**hum**" and "**windspeed**" variables

The total number of outliers for the continuous variables can be retrived as follows

```
# total outliers for each column
Q1 = data[numerical_vars].quantile(0.25)
Q3 = data[numerical_vars].quantile(0.75)
IQR = Q3 - Q1
((data[numerical_vars] < (Q1 - 1.5 * IQR)) | (data[numerical_vars] > (Q3 + 1.5 * IQR))).sum()
```

```
temp          0
atemp         0
hum           2
windspeed    13
dtype: int64
```

Clearly "**hum**" and "**windspeed**" have outliers. We can replace them with means of corresponding column. But before doing that, we have to replace the outliers with NA's

```
# Detect outliers but now replace them
for i in ["hum","windspeed"]:
    print(i)
    # calculate 75th and 25th percentile
    q75, q25 = np.percentile(data.loc[:,i] , [75,25])
    # calculate inter quartile range
    iqr = q75 - q25

    min = q25 - (iqr * 1.5)
    max = q75 + (iqr * 1.5)
    print(min)
    print(max)

    data.loc[data[i] < min , i] = np.nan
    data.loc[data[i] > max , i] = np.nan
```

Then we can replace NA's with means of corresponding columns using the **mean()** function

```
for i in ["hum","windspeed"]:

    data[i] = data[i].fillna(data[i].mean())
```

R code

To plot the boxplots in R, we require the ggplot2 library. First, we need to import it

```
library(ggplot2)
```

Then we plot the boxplot by the following code

```
numerical_variables = c("temp","atemp","hum","windspeed")

for(i in 1:length(numerical_variables)){
  assign(paste0("gn",i), ggplot(aes_string(y = (numerical_variables[i]), x = "cnt"), data = subset(data))+
           stat_boxplot(geom = "errorbar", width = 0.5) +
           geom_boxplot(outlier.colour="red", fill = "green" ,outlier.shape=15,
                        outlier.size=1, notch=FALSE) +
           theme(legend.position="bottom")+
           labs(y=numerical_variables[i],x="cnt")+
           ggtitle(paste("Box plot of cnt for",numerical_variables[i])))
}

# ## Plotting plots together
gridExtra::grid.arrange(gn1,gn2,ncol=2)

gridExtra::grid.arrange(gn3,gn4,ncol=2)
```

"**hum**" and "**windspeed**" have outliers which we need to replace by the means of corresponding columns. First we convert outliers to NA's and then impute them using the mean.

```
# #Replace all outliers with NA and impute
for(i in numerical_variables){
  val = data[,i][data[,i] %in% boxplot.stats(data[,i])$out]

  data[,i][data[,i] %in% val] = NA
}

# replacing NA's with means
for(i in cnames){

  data[,i][is.na(data[,i])] <- mean(data[,i], na.rm = TRUE)
}
```

2.1.4. Exploratory Data Analysis and Feature Selection

Exploratory data analysis is an approach to analysing data sets to summarize their main characteristics, often with visual methods. EDA is for seeing what the data can tell us beyond the formal modeling or hypothesis testing task. Based on findings of EDA, we select the relevant features

EDA on Continuous Variables:

First, let us visualise the heatmap of correlation matrix between the continuous variables. We later move on to visualize scatter plots along with histograms.

Python code for EDA on continuous variables

We plot the heatmap of correlation matrix for numeric variables using the **matplotlib** and **seaborn** libraries.

```
import matplotlib.pyplot as plt
import seaborn as sn
```

```
# Correlation analysis
# correlation plot
df_corr = data.loc[:,numerical_vars]
```

```
%matplotlib inline
# Set the width and height of the plot
f , ax = plt.subplots(figsize = (20,10))

# Generate correlation matrix
corr = df_corr.corr()

# Plot using seaborn library
sn.heatmap(corr,mask = np.zeros_like(corr,dtype = np.bool), cmap = sn.diverging_palette(220,10,as_cmap = True),
           square = True , ax = ax , vmin = -1 , vmax = 1 , annot = True)
```

Following is the correlation plot is generated

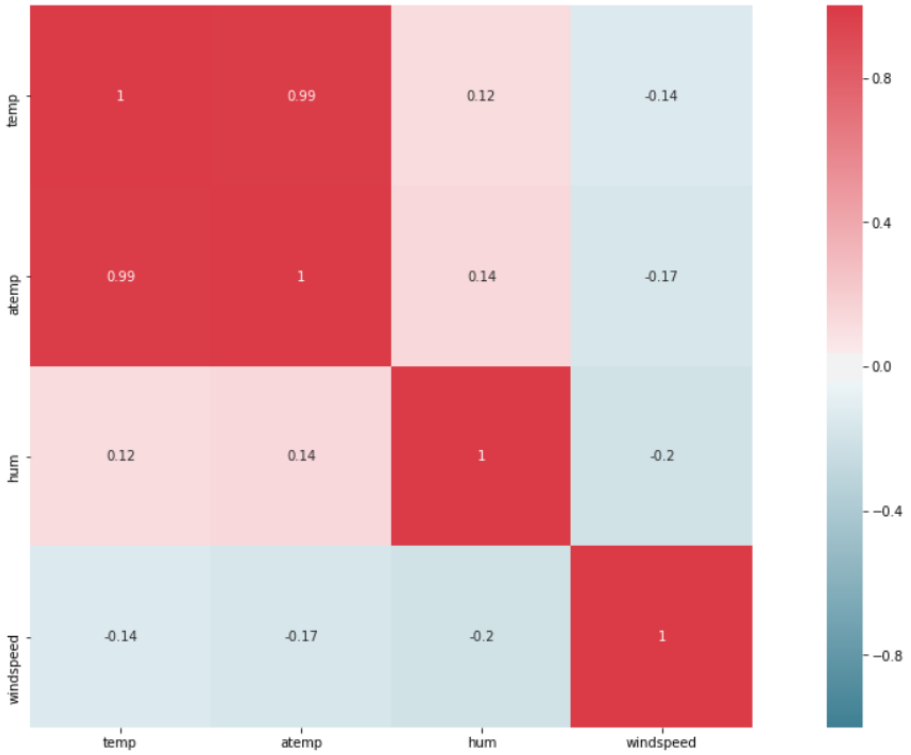

Fig 3: Correlation Plot for Continuous Variables

Clearly "**temp**" and "**atemp**" are **highly correlated**. This can affect our model negatively. Hence we have to decide on which variable to be removed. This can be achieved by comparing their **Variance Inflation Factors.** We compare the **VIF's** of the numerical variables as follows

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
In [11]: df = data[numerical_vars]

In [12]: X = add_constant(df)

In [13]: pd.Series([variance_inflation_factor(X.values,i)
                    for i in range(X.shape[1])],
                   index = X.columns)

Out[13]: const         45.499530
         temp          63.010048
         atemp         63.632085
         hum            1.059230
         windspeed      1.097383
         dtype: float64
```

We see that "**atemp**" has higher **Variance Inflation Factor** (VIF). This variable can be **removed** later.

Let us visualize scatter plots of the continuous variables versus dependent variable '**cnt**'

Following is the code to generate scatter plots for the continuous variables

```
# scatter plots of continuous variables vs cnt
for col in numerical_vars:
    sn.scatterplot(x = col, y = 'cnt' , data = data)
    plt.show()
```

Following are the scatter plots generated

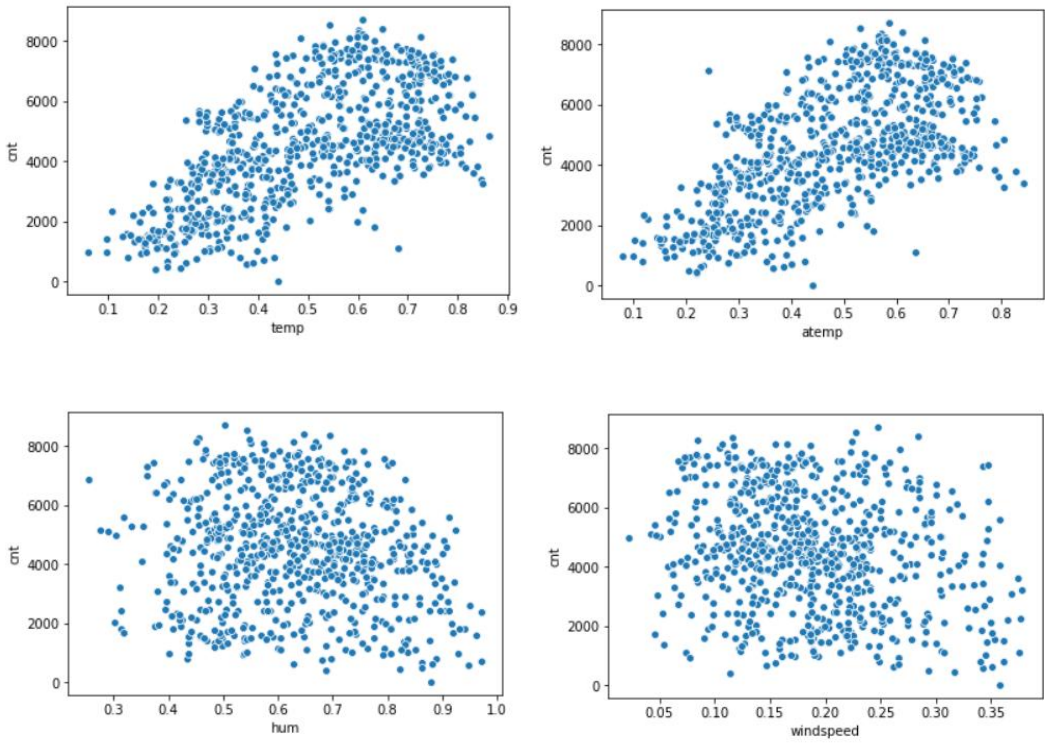

Fig 4: Scatter plot of continuous variables versus dependent variable '**cnt**'

We can see that, "**temp**" and "**atemp**" are highly correlated with dependent variable "**cnt**"

Next we visualize the histogram to check if the continuous variables have a normal distribution

```
# Plotting histograms of continuous variables

for col in numerical_vars:

    plt.hist(data[col],bins = 'auto' ,ec='black')
    plt.xlabel(col)
    plt.show()
```

Following are the histograms generated

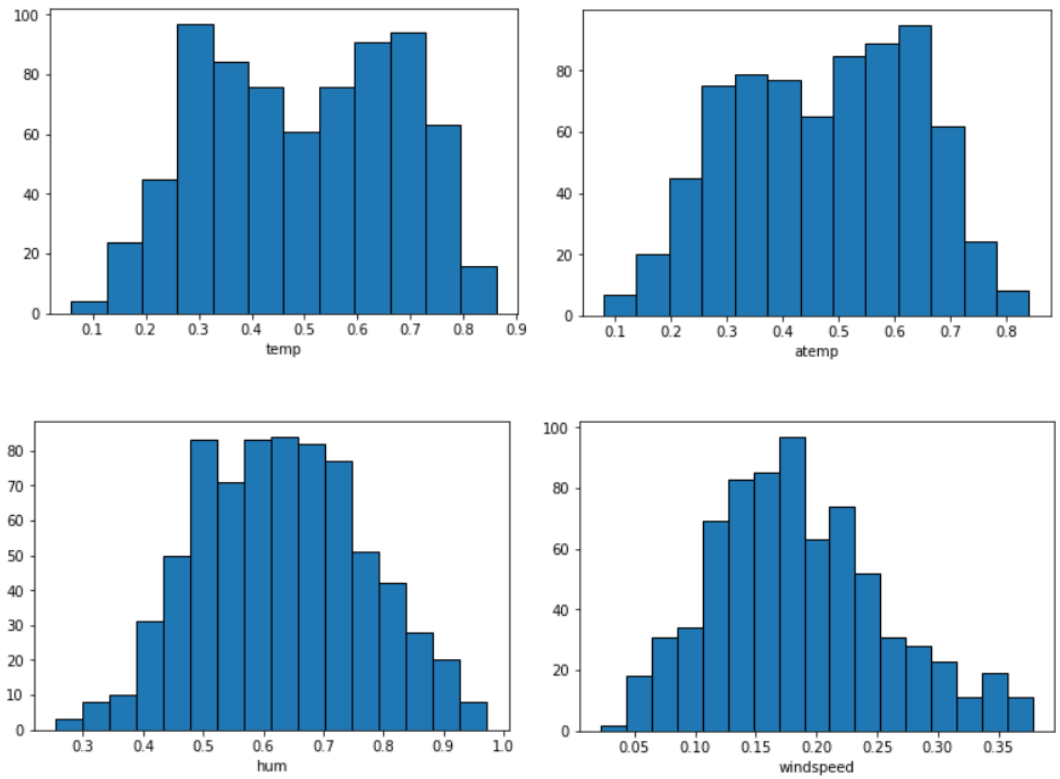

Fig 5: Histograms for continuous variables

It can be clearly seen the numerical variables do not have a normal distribution.

R code for EDA on continuous variables

The Correlation plot can be plotted as follows using the **corrgram** library in R

```
library(corrgram)
```

By doing the following we can plot the correlation plot

```
corrgram(data[numerical_variables], order=FALSE,
         main="correlation plot",upper.panel=panel.pie,text.panel=panel.txt)
```

After plotting the correlation matrix, we see that "**temp**" is highly correlated with "**atemp**". We have to remove one of the variables since this **multicollinearity** will negatively impact the model.

We have to compare their Variance Inflation factors in order to decide which variable to be dropped.

To compute VIF's we require the **HH** library.

```
library(HH)
```

Next we compute VIF's for the numerical variables using **vif()** function

```
> print(vif(data[,c(8,9,10,11)]))
     temp     atemp       hum windspeed
63.010048 63.632085  1.059230  1.097383
```

Clearly "**atemp**" has higher **VIF** compared to '**temp**'. Later during conclusion of EDA on continuous variables, we will drop '**atemp**'

Let's visualize the relation between the numerical variables and the dependent variable by plotting the **scatter plots** using the following code.

```
scatterplotlist = list()

for(i in numerical_variables){
  plot = ggplot(data, aes_string(x = data[,i] , y = data$cnt)) +
         geom_point() +
         theme_bw() + ylab("cnt") + xlab(i) + ggtitle("Scatter Plot Analysis") +
         theme(text = element_text(size = 15))+
         scale_x_continuous(breaks = pretty_breaks(n = 10)) +
         scale_y_continuous(breaks =  pretty_breaks(n = 10))

  t  = length(scatterplotlist) + 1
  scatterplotlist[[t]] = plot

}
grid.arrange(grobs = scatterplotlist)
```

Through the scatter plot, we see that "**atemp**" and "**temp**" are highly correlated with "**cnt**".

Next, we visualize the distribution of the continuous variables using **histograms** using the following code.

```
histogramlist = list()


for(i in numerical_variables){
  plot = ggplot(data , aes_string( x = data[,i])) +
    geom_histogram(fill = "cornsilk" , colour = "black") + geom_density() +
    scale_y_continuous(breaks = pretty_breaks(n = 10)) +
    scale_x_continuous(breaks = pretty_breaks(n = 10)) +
    theme_bw() + xlab(i) + ylab("Frequency") +
    ggtitle(i) + theme(text = element_text(size = 15))


  t  = length(histogramlist) + 1
  histogramlist[[t]] = plot

}
grid.arrange(grobs = histogramlist)
```

We see that the continuous variables are not normally distributed.

Conclusion after EDA on continuous variables

After preforming EDA on continuous variables both in python and R, we conclude that "**atemp**" will have **negative impact** on the model. We have to remove the "**atemp**" variable. We also conclude that since there is no multivariate normality, Linear Regression technique won't be effective on the dataset.

Finally we drop "**atemp**" variable

Python code to drop "atemp" variable

```
data = data.drop(columns = ["atemp"])
```

R code to drop "atemp" variable

```
data$atemp = NULL
```

Next, we will start with exploratory data analysis on **categorical variables**.

Exploratory Data Analysis on Categorical Variables

We need to understand the relation between the categorical variables and the dependent variable '**cnt**'. This can be done using **bar plots** and **ANOVA** which uses **F-test**

Python code for EDA on categorical variables:

We already have our categorical variables in the label encoded format in the data set. The categorical variables are "**season**","**yr**","**mnth**","**holiday**","**weekday**","**workingday**","**weathersit**"

Let us plot a bar plot of all the categorical variables versus dependent variable '**cnt**'. Using the **barplot()** function from the **seaborn** library, we can plot the bar graphs.

```
category_names = ["season","yr","mnth","holiday","weekday","workingday","weathersit"]
```

```
for category in category_names:
    sn.set(style = "whitegrid")
    sn.barplot(x = category, y = 'cnt' , data = data)
    plt.show()
```

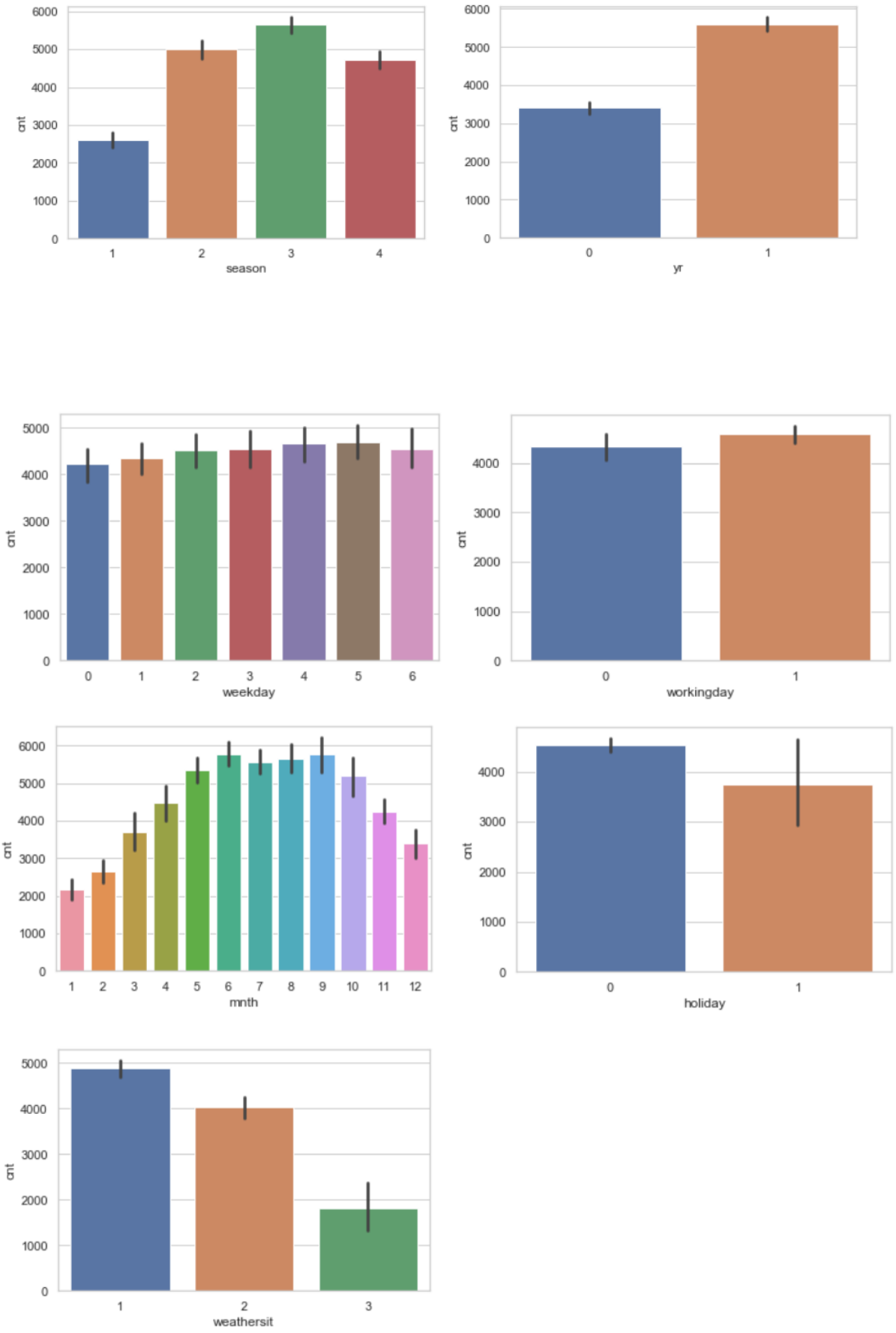

Fig 7: Bar Plot to visualize the relationship between categorical variables and the dependent variable '**cnt**'

There are many conclusions which we can derive from the bar plot of these categorical variables versus the dependent variable "**cnt**".

We see that the average bike rental count is maximum during fall and minimum during spring

More bikes were rented in 2012 as compared to 2011

More bikes were rented in September and less number of bikes during January

More bikes were rented when there were no holidays

More bikes were rented on Saturdays and less on Monday

If the day is neither weekend nor holiday, more bikes were rented

Clear weather along with few clouds increase the bike rental count

Next we apply **ANOVA** using **F-test** to rank the categorical features based on their ability to explain the variance in the target variable "**cnt**".

In the code below, we apply **ANOVA** to rank the categorical features based on their role in determining the target variable

In [146]:
```
##### Applying ANOVA to understand which categorical variable impact rental count the most
##### F test for selecting categorical variables
##### we will select 7 columns having best F-values

from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
import sklearn.feature_selection as fs

temp_df = data[category_names]

kb = fs.SelectKBest(f_regression,k=7)
kb.fit(temp_df, data['cnt'])
names = temp_df.columns.values[kb.get_support()]
scores = kb.scores_[kb.get_support()]
names_scores = list(zip(names, scores , kb.pvalues_ ))
ns_df = pd.DataFrame(data = names_scores, columns=['Feat_names','F_Scores', 'P_Vals'])

ns_df_sorted = ns_df.sort_values(['F_Scores','Feat_names'], ascending =[False, True])

ns_df_sorted
```

Out[146]:

	Feat_names	F_Scores	P_Vals
1	yr	344.890586	2.483540e-63
0	season	143.967653	2.133997e-30
6	weathersit	70.729298	2.150976e-16
2	mnth	62.004625	1.243112e-14
3	holiday	3.421441	6.475936e-02
4	weekday	3.331091	6.839081e-02
5	workingday	2.736742	9.849496e-02

We see that "**yr**" has highest **F statistic** value and is the top most features to explain variance in '**cnt**'. On the other hand '**workingday**' has the least **F statistic** value

Following code is to plot a bar graph between categorical variables and their respective F values

```
# F-value can be used for feature ranking
# Barplot of f-values for different features
sn.barplot(data = ns_df_sorted , x = "Feat_names", y = "F_Scores")
```

The above code gives us the following bar plot

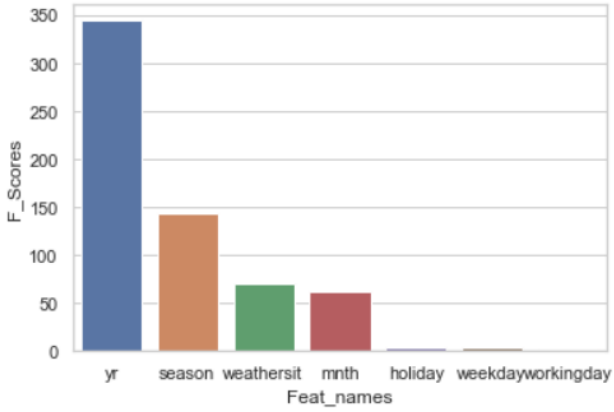

Fig 8: Bar plot of categorical variables versus F scores

R code for EDA on categorical variables

Following is the code in R to plot the bar plots of different categorical variables versus the dependent variable count.

```
# bar plot of categorical features versus dependent variable cnt
categorical_vars = c("season","yr","mnth","holiday","weekday","workingday","weathersit")

barplotlist = list()

for(i in categorical_vars){


  plot = ggplot(data , aes_string( x = as.factor(data[,i]) , y = data$cnt)) +
    geom_bar(stat = "summary" , fill = "DarkSlateBlue" , fun.y =mean) + theme_bw() +
    xlab(i) + ylab("cnt") + scale_y_continuous(breaks = pretty_breaks(n = 8)) +
    ggtitle("Bike rental analysis") + theme(text = element_text(size = 15))

  t  = length(barplotlist) + 1
  barplotlist[[t]] = plot

}

grid.arrange(barplotlist[[1]], barplotlist[[2]])
grid.arrange(barplotlist[[3]], barplotlist[[4]])
grid.arrange(barplotlist[[5]], barplotlist[[6]])
grid.arrange(barplotlist[[7]])
```

Next, we apply ANOVA. Following is the code to get **F Statistic** values for different categories. We require the **aov()** function to compute **F statistic** values. We also create a data frame which stores the categorical variables with corresponding **F statistic** values. We also plot the bar graph of categorical variables versus the **F statistic**

```
## anova
categorical_vars = c("season","yr","mnth","holiday","weekday","workingday","weathersit")

# ANOVA
Fvalue_list = c()
k = 1
df = data.frame( matrix( ncol = 2 , nrow = 7))

x = c("categorical_variable","F_value")
colnames(df) = x

for(i in categorical_vars){

  a = aov(data[,i] ~ data$cnt , data = data)
  t = summary(a)
  Fvalue_list[[k]] = t[[1]]$F[[1]]
  k = k+1
}

df$categorical_variable = categorical_vars
df$F_value = Fvalue_list

ggplot(df,aes(x= reorder(df$categorical_variable,-df$F_value),df$F_value))+geom_bar(stat ="identity")+
  xlab("categorical_variable") + ylab("Fvalue")
```

From the bar plot we conclude that "**yr**" is has the highest and "**workingday**" has the lowest **F statistic** value

2.2. Model Development and Evaluation

2.2.1. Model Selection

We have to predict the count of bike rental. Our target variable "**cnt**" is a continuous variable. Clearly this is a regression problem. The model which will give the lowest **mean absolute error** (**MAE**) will be our final model. We have used the following Regression algorithms

1. Decision Tree Regression

2. Random Forest Regression

3. Gradient Boost Regression

Before applying the model we have to divide the dataset in **training** and **testing** dataset. We are considering 75% of the data for training and 25% for testing purpose.

Python code

In python we need to import **train_test_split()** function from the **scikit learn** (**sklearn**) library to split the data into training and testing data sets

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(data.drop('cnt',axis=1),data['cnt'],test_size=0.25,random_state=2)
```

We have also created a custom function to calculate Mean Absolute Percentage error.

```
# Calculate MAPE
def MAPE(y_true, y_pred):
    mape = np.mean(np.abs((y_true-y_pred)/y_true))
    return mape
```

We also import **mean_squared_error()** and **mean_absolute_error()** from **sklearn** to compute Mean Squared Error and Mean Absolute Error respectively.

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

R code

In R, using the **sample()** function we can split the data into training and testing data.

```
set.seed(1234)
# Divide the data set into train and test
train_index = sample(1:nrow(data) , 0.75 * nrow(data))

train = data[train_index,]
test = data[-train_index,]
```

2.2.2. Decision Tree Regression

Python code

To apply decision tree regression, we need to import **DecisionTreeRegressor** from **sklearn**

```
from sklearn.tree import DecisionTreeRegressor
```

Then we build the model and predict count values for the test data

```
dtree_model = DecisionTreeRegressor(random_state=42)
dtree_model.fit(x_train,y_train)
dtree_predictions = dtree_model.predict(x_test)
```

Following are the model evaluation metrics we obtain

```
print("MAE for decision tree regressor is ",mean_absolute_error(y_test,dtree_predictions))
print("MAPE for decision tree regressor is ",MAPE(y_test,dtree_predictions))
print("MSE for decision tree regressor is ",mean_squared_error(y_test,dtree_predictions))
print("RMSE for decision tree regressor is ",np.sqrt(mean_squared_error(y_test,dtree_predictions)))
```

```
MAE for decision tree regressor is  624.6448087431694
MAPE for decision tree regressor is  0.18283266755725586
MSE for decision tree regressor is  699043.8032786886
RMSE for decision tree regressor is  836.0883944169352
```

Using **ANOVA**, we had found out that "**workingday**" has the lowest rank in explaining the variable in the target variable '**cnt**'. Still by dropping the low ranked features, there wasn't any improvement in our metrics.

Further, we apply **hyperparameter tuning** to improve on our metrics. We have considered different combinations of **min_sample_leaf** and **max_features.** Here **min_sample_leaf** is the minimum number of samples required to be at a leaf node and **max_features** are the number of features to be considered when looking for the best split. We are performing a 3-fold cross validation and finding the best combination of parameters for which the **MAE** value is the least. Using the optimal parameters we make the predictions. We use the **GridSearchCV()** function from **sklearn** which does an exhaustive search over specified parameter values for an estimator. Its important features are **fit** and **predict**

```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
```

```
kf = KFold(n_splits= 3 , random_state=42)
params_dict = {
              'min_samples_leaf': list(range(1,20,2)),
              'max_features': list(range(1,11))
              }

dtree_tune=GridSearchCV(estimator=DecisionTreeRegressor(random_state=42),
                        param_grid=params_dict ,
                        cv = kf,
                        scoring='neg_mean_absolute_error',
                        verbose=6000)
dtree_tune.fit(x_train,y_train)
```

Following is the best estimator found through grid search

```
dtree_tune.best_estimator_
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=10,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=5,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=42, splitter='best')
```

Next we make the predictions and obtain by applying the model on our test data

```
dtree_tuned_predictions=dtree_tune.predict(x_test)

print("MAE after decision tree after hyperparameter tuning is ",mean_absolute_error(y_test,dtree_tuned_predictions))
print("MAPE after decision tree after hyperparameter tuning is ",MAPE(y_test,dtree_tuned_predictions))
print("MSE after decision tree after hyperparameter tuning is ",mean_squared_error(y_test,dtree_tuned_predictions))
print("RMSE after decision tree after hyperparameter tuning is ",
      np.sqrt(mean_squared_error(y_test,dtree_tuned_predictions)))
```

```
MAE after decision tree after hyperparameter tuning is  583.1435467083008
MAPE after decision tree after hyperparameter tuning is  0.17279279436900477
MSE after decision tree after hyperparameter tuning is  640628.326230909
RMSE after decision tree after hyperparameter tuning is  800.3926075563847
```

We see that there is a significant improvement in **Mean absolute error** along with other metrics.

R code for Decision tree regression

We have considered CART (decision tree regression) algorithm. For that we have to have to use **rpart** package.

```
library(rpart)
```

Next, we build the model using the **rpart()** function and predict for test cases using **predict()** function

```
set.seed(1234)

fit = rpart(cnt ~. , data = train , method = "anova")

# Predict for new test cases
predictions = predict(fit,test[,-11])
```

We use the **eval()** function from **DMwR** library to obtain the evaluation metrics

```
library(DMwR)

> regr.eval(test[,11], predictions , stats = c('mae','rmse','mape','mse'))
         mae         rmse         mape          mse
6.465874e+02 8.658923e+02 2.379297e-01 7.497695e+05
```

We plot the decision tree generated

```
rpart.plot::rpart.plot(fit)
```

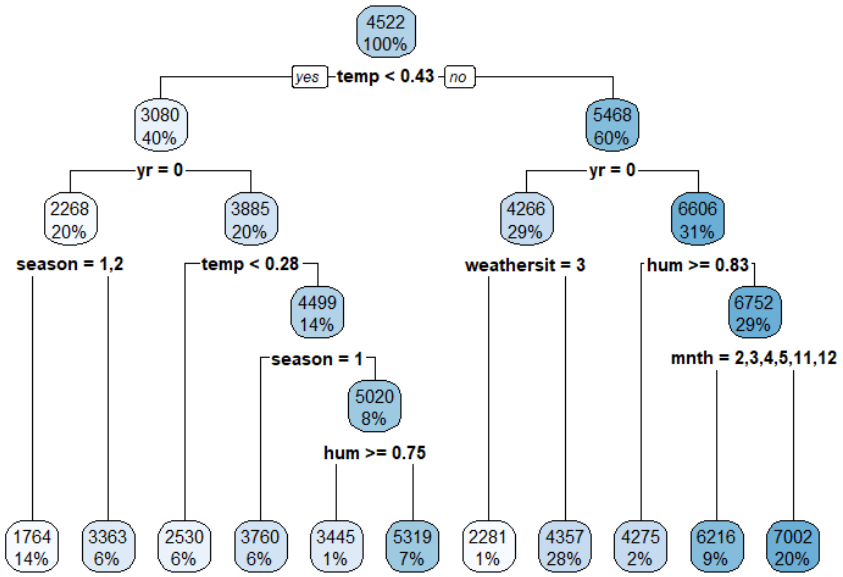

Fig 9: Decision tree generated using rpart

Next, we perform **hyperparameter tuning** by using the **caret** package in R. Method "**rpart**" is capable of only tuning **cp** parameter. Here **cp** is the **complexity parameter** used by **rpart** to determine when to prune the tree. The **caret** package contains **train()** function which is used to fit predictive models over different tuning parameters. The **trainControl()** function control the parameters for train.

```
# tuning rpart
library(caret)
rpart_grid = expand.grid(cp = c(0.1,0.2,0.01,0.02,0.001,0.002))

# Define the control
trControl <- trainControl(method = "cv", number = 3, search = "grid",
                          verboseIter = TRUE)

set.seed(1234)

tuned_tree_model = caret::train(cnt ~ .,
                                data = train,
                                method = "rpart",
                                metric = "MAE",
                                trControl = trControl,
                                tuneGrid = rpart_grid)
```

We get the optimal **cp** value as 0.001

```
> tuned_tree_model$bestTune$cp
[1] 0.001
```

Now taking the optimal parameters, we make the prediction for the count variable and obtain the metrics

```
predictions = predict(tuned_tree_model,test[,-11])
```

```
> regr.eval(test[,11], predictions , stats = c('mae','rmse','mape','mse'))
         mae         rmse         mape          mse
5.888964e+02 7.660456e+02 2.212412e-01 5.868259e+05
```

Here we see a significant improvement in **Mean Absolute Error** along with in the metrics

2.2.3. Random Forest Regression

Python code

To apply Random Forest Regression, we need to import **RandomForestRegressor** from **sklearn**

```
from sklearn.ensemble import RandomForestRegressor
```

Then we build the model and predict the bike rental count for the test data

```
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(x_train,y_train)
rf_predictions = rf_model.predict(x_test)
```

Following are the model evaluation metrics which we obtain

```
print("MAE for random forest regressor is ",mean_absolute_error(y_test,rf_predictions))
print("MAPE for random forest regressor is ",MAPE(y_test,rf_predictions))
print("MSE for random forest regressor is ",mean_squared_error(y_test,rf_predictions))
print("RMSE for random forest regressor is ",np.sqrt(mean_squared_error(y_test,rf_predictions)))
```

```
MAE for random forest regressor is  509.5945355191257
MAPE for random forest regressor is  0.1591791117864434
MSE for random forest regressor is  488724.9433879781
RMSE for random forest regressor is  699.0886520234599
```

Dropping some of the lowest ranked features found out through ANOVA, still didn't improve our metrics.

Next we go for **hyperparameter optimization** to tune our model and improve the evaluation metrics. We have considered different combinations of **n_estimators** , **max_features** and **max_depth**. Here **n_estimators** are the number of trees in the forest, **max_features** are the number of features to be considered when looking for the best split, **max_depth** is the maximum depth of the tree. We are performing 3-fold cross validation and finding the optimal parameters that result in least **MAE**.

```
kf = KFold(n_splits= 3 , random_state=42)

param_grid={'n_estimators':[100,200,300,400,500],'max_features':list(range(1,11)),'max_depth': [4,6,8]}

rf_tune=GridSearchCV(RandomForestRegressor(random_state=42),
                     param_grid=param_grid ,
                     cv=kf ,
                     scoring = 'neg_mean_absolute_error',
                     verbose= 6000)

rf_tune.fit(x_train,y_train)
```

Following is the best estimator found through the grid search

```
rf_tune.best_estimator_
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=8,
                      max_features=6, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=200,
                      n_jobs=None, oob_score=False, random_state=42, verbose=0,
                      warm_start=False)
```

Next we make predictions and obtain evaluation metrics

```
rf_tuned_predictions=rf_tune.predict(x_test)

print("MAPE after random forest hyperparameter tuning is ",MAPE(y_test,rf_tuned_predictions))
print("MAE after random forest hyperparameter tuning is ",mean_absolute_error(y_test,rf_tuned_predictions))
print("MSE after random forest hyperparameter tuning is ",mean_squared_error(y_test,rf_tuned_predictions))
print("RMSE after random forest hyperparameter tuning is ",np.sqrt(mean_squared_error(y_test,rf_tuned_predictions)))
```

```
MAPE after random forest hyperparameter tuning is  0.14447484271238206
MAE after random forest hyperparameter tuning is  474.4333269936359
MSE after random forest hyperparameter tuning is  422038.1917541686
RMSE after random forest hyperparameter tuning is  649.6446657628835
```

We see a significant improvement in **Mean Absolute Error** and other evaluation metrics

R code for Random Forest Regression

We use the **randomForest()** function from **randomForest** library and make predictions using **predict()** function

```
library(randomForest)

set.seed(1234)
rf_model = randomForest(cnt~.,data = train)

# Predict for new test cases
rf_predictions = predict(rf_model,test[,-11])
```

We get the following evaluation metrics

```
> regr.eval(test[,11], rf_predictions , stats = c('mae','rmse','mape','mse'))
         mae         rmse         mape          mse
4.624594e+02 6.179459e+02 1.787882e-01 3.818572e+05
```

Further, we do **hyperparameter tuning** using the caret package. We experiment with different **mtry** values and select the one resulting in least MAE. Here **mtry** is the number of variables randomly sampled as a candidate at each split node. We perform a repeated 10-fold cross validation 3 times and find the optimal parameters.

```
set.seed(1234)

# default ntrees = 500
rf_tuned_model = caret::train(cnt~.,
                              data = train,
                              method = 'rf',
                              metric = 'MAE',
                              trControl = trainControl(method = "repeatedcv",
                                                       number = 10,
                                                       repeats = 3,
                                                       verboseIter = TRUE
                                                       ),
                              tuneGrid = expand.grid(.mtry = c(1:10)),
                              ntree = 500
                              )
```

We get the following optimal parameter

```
rf_tuned_model$bestTune
mtry
   6
```

Next, we make the predictions

```
rf_tuned_predictions = predict(rf_tuned_model,test[,-11])
```

We get the following evaluation metrics

```
> regr.eval(test[,11], rf_tuned_predictions , stats = c('mae','rmse','mape','mse'))
          mae          rmse          mape           mse
4.277335e+02 5.890596e+02 1.615362e-01 3.469912e+05
```

Here too there a significant improvement in **Mean Absolute Error** and other evaluation metrics

2.2.4. Gradient Boost Regression

Python code for Gradient Boosting Regression:

To perform Gradient Boosting Regression, we have to use **GradientBoostingRegressor** from **sklearn**.

```
from sklearn.ensemble import GradientBoostingRegressor
```

Then we build the model and predict the bike rental count for our test data

```
gbr_model = GradientBoostingRegressor(random_state=42)
gbr_model.fit(x_train,y_train)
test_pred = gbr_model.predict(x_test)
```

Following are the model evaluation metrics which we obtain

```
print("MAE for gradient boosting regressor is ",mean_absolute_error(y_test,gbr_predictions))
print("MAPE for gradient boosting regressor is ",MAPE(y_test,gbr_predictions))
print("MSE for gradient boosting regressor is ",mean_squared_error(y_test,gbr_predictions))
print("RMSE for gradient boosting regressor is ",np.sqrt(mean_squared_error(y_test,gbr_predictions)))
```

```
MAE for gradient boosting regressor is  495.4848806408682
MAPE for gradient boosting regressor is  0.14976151863699272
MSE for gradient boosting regressor is  433500.6448948831
RMSE for gradient boosting regressor is  658.407658593734
```

Next, we perform **hyperparameter tuning** to improve our metrics. We consider different combinations of **learning_rate**, **max_depth**, **min_samples_leaf**, **n_estimators**. The **learning_rate** indicates the learning rate, **max_depth** the maximum depth of individual regression estimators and **n_estimators** indicates the number of boosting stages to perform. We are performing 3-fold cross validation to determine the optimal parameters that result in least MAE

```
kf = KFold(n_splits= 3 , random_state=42)

gb_grid_params = {'learning_rate': [0.1, 0.05, 0.02, 0.01],
              'max_depth': [4, 6, 8],
              'min_samples_leaf': [20,30,40],
              'n_estimators' : [1000,2000,3000,4000,5000]
              }

gbr_tuned = GridSearchCV(GradientBoostingRegressor(random_state=42),
                  gb_grid_params,
                  cv=kf,
                  scoring = "neg_mean_absolute_error",
                  verbose = 6000)

gbr_tuned.fit(x_train,y_train)
```

We get the following best estimator

```
gbr_tuned.best_estimator_
```

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.02, loss='ls', max_depth=4,
                          max_features=None, max_leaf_nodes=None,
                          min_impurity_decrease=0.0, min_impurity_split=None,
                          min_samples_leaf=20, min_samples_split=2,
                          min_weight_fraction_leaf=0.0, n_estimators=1000,
                          n_iter_no_change=None, presort='auto',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

Now we make the predictions and obtain the evaluation metrics

```
gbr_tuned_predictions=gbr_tuned.predict(x_test)

print("MAPE after GBR hyperparameter tuning is ",MAPE(y_test,gbr_tuned_predictions))
print("MAE after GBR hyperparameter tuning is ",mean_absolute_error(y_test,gbr_tuned_predictions))
print("MSE after GBR hyperparameter tuning is ",mean_squared_error(y_test,gbr_tuned_predictions))
print("RMSE after GBR hyperparameter tuning is ",np.sqrt(mean_squared_error(y_test,gbr_tuned_predictions)))
```

```
MAPE after random forest hyperparameter tuning is  0.1444996791682019
MAE after random forest hyperparameter tuning is  458.24667329990666
MSE after random forest hyperparameter tuning is  413726.5343671407
RMSE after random forest hyperparameter tuning is  643.21577590039
```

We see that **Mean Absolute Error** and other metrics have improved significantly

R code for Gradient Boosting Regression

We are making use of **caret** package for tuning the parameters in Gradient Boost Regression. We have used different combinations **of interaction.depth** , **n.trees** , **shrinkage** and **n.minobsinnode.** Here **interaction.depth** indicates maximum depth of each tree, **n.trees** indicates total number of trees to fit, **shrinkage** is the learning rate, **n.minobsinnode** indicates number of observations in the terminal node. We are performing 3-fold cross validation on our training data.

```
gbmGrid = expand.grid(interaction.depth = c(4,6,8),
                      n.trees =(1:40)*50,
                      shrinkage = c(0.1,0.01) ,
                      n.minobsinnode = c(10,20))

# Define the control
trControl <- trainControl(method = "cv", number = 3, search = "grid",
                          verboseIter = TRUE)

set.seed(1234)
gbr_tuned_model = caret::train(cnt ~ .,data = train, method = "gbm", metric = "MAE",
                     trControl = trControl, tuneGrid = gbmGrid ,
                      verbose = FALSE)
```

We get the following optimal parameters.

```
> gbr_tuned_model$bestTune
    n.trees interaction.depth shrinkage n.minobsinnode
180    1000                 8      0.01             10
```

Now we make the predictions and compute the evaluation metrics

```
gbr_tuned_predictions = predict(gbr_tuned_model,test[,-11])
```

```
> regr.eval(test[,11], gbr_tuned_predictions , stats = c('mae','rmse','mape','mse'))
          mae          rmse          mape           mse
4.213192e+02 5.706565e+02 1.429502e-01 3.256489e+05
```

Chapter 3

Conclusion

3.1. Final Best Model Selection

For final Model Selection, we will choose the model which gives us the lowest **Mean Absolute Error** value. After applying the entire regression model, we observed that **Gradient Boosting Regression** performs the best and results in lowest **Mean Absolute Error**. We will make use of **Gradient Boosting Regression** in both python and R to make predictions on the test data set.

Python

The following tables summarizes the results obtained by different regression models in python

Regression Model	MAE	MAPE	MSE	RMSE
Decision Tree Regression	583.1435	0.1728	640628.3262	800.3926
Random Forest Regression	474.4333	0.1445	422038.1918	649.6447
Gradient Boosting Regression	458.2467	0.1445	413726.5347	643.2158

We observe that Gradient Boost Regression achieves the lowest **Mean Absolute Error**. Next, we store the predictions we had made using Gradient Boosting Regression in a .csv file

Following is the code to store results in a .csv file.

```
# creating instant so that we can submit it to csv along with predicted count
# instant is 1 greater than row index

submission_df = pd.DataFrame()
submission_df["instant"] = x_test.index + 1
```

```
## submitting Gradient Boost predictions to csv file
#Create a  DataFrame with instants and their predicted counts and store results to csv
output = pd.DataFrame({'instant':submission_df["instant"],'predicted count':gbr_tuned_predictions.round()})

#Convert DataFrame to a csv file that can be uploaded
#The predictions are saved to a csv file
file = 'Predictions using Python.csv'

output.to_csv(file,index=False)

print('csv file:' , file, ' with predicted count values has been genrated successfully')
```

```
csv file: Predictions using Python.csv  with predicted count values has been genrated successfully
```

R programming

The following tables summarizes the results obtained by different regression models in R

Regression model	MAE	MAPE	MSE	RMSE
Decision Tree Regression	588.8964	0.2212	586825.9	766.0456
Random Forest Regression	427.7335	0.1615	346991.2	589.0596
Gradient Boosting regression	421.3192	0.1429	325648.9	570.6565

We observe that **Gradient Boost Regression** achieves the lowest **Mean Absolute Error** among the three models. Next, we store the predictions we have made using Gradient Boosting Regression in a .csv file

```
#store results of gradient boost regression to csv
df = data.frame( matrix( ncol = 2 , nrow = nrow(test)))
x = c("instant","count_predictions")
colnames(df) = x

df$instant = as.numeric(rownames(test))
df$count_predictions = round(gbr_tuned_predictions)

file = "Predictions using R.csv"

write.csv(df,file, row.names = FALSE)

print(paste0("csv file ",file," with predicted count values has been generated successfully"))
```