PROJECT 2

BIKE RENTING

CONTENTS

Chapter 1

INTRODUCTION:-

FELTON,California, March 21,2019 /PRNewswire/ -- The global **bike rental market** is expected to reach **USD 4.00 billion by 2025**. Increasing traffic, easy access, attractive pricing, and support of local authorities are among the key driving factors driving the growth of bike rental services. Leveraging of digital platforms by service providers is also expected to add to the growing adoption of rental services in the coming years.

1.2 IN INDIA:- India a country with 1.21 billion population, with 60% living in metros. Dreams of riding premium bikes for long drives or adventure road trips or photo shoot or romantic dating are no more dreams in India. Easily available and affordable rental bikes have made it possible. More and more bike rental startups are coming up with an exclusive range of bikes for short errands, day-long drives and long trips across India and beyond. Cars and bikes provides you door to door services but Cars are not so affordable and consumes more space as compare to bikes as bikes or any other two-wheelers are very convenient as they consume very less space and can manage to move in traffic faster than cars.

Bike rental platforms operational in india:

Chapter 2

BIKE RENTAL DATA

2.1 PROBLEM STATEMENT -The objective of this Case is to Predication of bike rental count on daily based on the environmental and seasonal settings.

2.2 Variables Involved:-

The details of data 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 to 12)

hr: Hour (0 to 23)

holiday : weather day is holiday or not (extracted from Holiday Schedule)

weekday: Day of the week

working day : If day is neither weekend nor holiday is 1, otherwise is 0.

Weathersit : (extracted from Free meteo)

1: Clear, Few clouds, Partly cloudy, 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), t_min=-8, 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), t_min=-16, t_max=+50 (only in hourly scale)

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

windspeed: Normalized wind speed. The values are divided to 67 (max)

casual: count of casual users

registered: count of registered users

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

```
day.dtypes
```

```
instant         int64
dteday         object
season          int64
yr              int64
mnth            int64
holiday         int64
weekday         int64
workingday      int64
weathersit      int64
temp          float64
atemp         float64
hum           float64
windspeed     float64
casual          int64
registered      int64
cnt             int64
dtype: object
```

Chapter 3

DATA PRE-PROCESSING:

In any Machine Learning process, Data Preprocessing is that step in which the data gets transformed, or *Encoded*, to bring it to such a state that now the machine can easily parse it. In other words, the *features* of the data can now be easily interpreted by the algorithm.

3.1 Data exploration, Missing Values and Outlier analysis:

Firstly, we perform data exploration and cleaning which includes following points as per this project:

Convert the data types into appropriate data types.

```
### data pre processing and data exploration
## converting the data into required data types
day['dteday'] = pd.to_datetime(day['dteday'],yearfirst = True)
day['season'] = day['season'].astype(str)
day['yr']     = day['yr'].astype(str)
day['mnth']   = day['mnth'].astype(str)
day['holiday']= day['holiday'].astype(str)
day['weekday']= day['weekday'].astype(str)
day['workingday']= day['workingday'].astype(str)
day['weathersit']=day['weathersit'].astype(str)
```

```
day.dtypes
```

```
instant                int64
dteday        datetime64[ns]
season                object
yr                    object
mnth                  object
holiday               object
weekday               object
workingday            object
weathersit            object
temp                 float64
atemp                float64
hum                  float64
windspeed            float64
casual                 int64
registered             int64
cnt                    int64
dtype: object
```

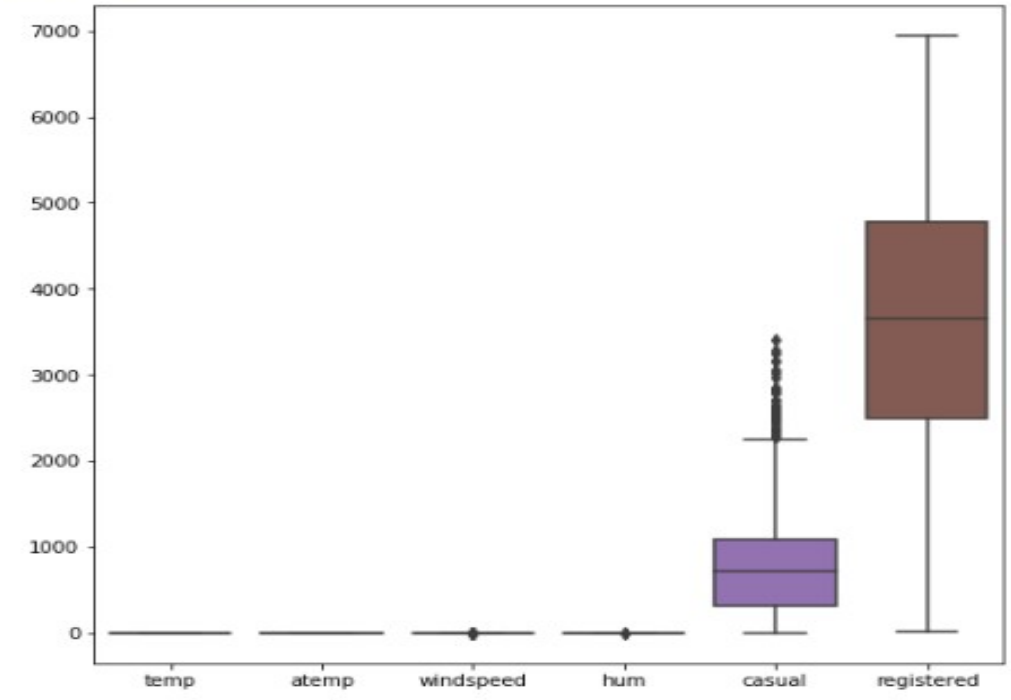
Check the missing values in the data.

```
In [10]: day.isnull().sum()  ### missing values

Out[10]: 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
```

3.2 Box plot analysis:- it the necessary for outlier analysis, it is mainly performed to analyze the outlier values present in the given dataset. Boxplots are a standardized way of displaying the distribution of data based on a five number summary ("minimum", first quartile (Q1), median, third quartile (Q3), and "maximum").

```
%matplotlib inline
sns.boxplot(data=day[['temp','atemp','windspeed','hum','casual','registered']])
fig=plt.gcf()
fig.set_size_inches(8,8)
```



from box plot analysis it was found that windspeed and hum (humidity) had few outliers. Here we do not consider outliers in casual because that won't help in building the model, not a predictor variable, sum of 'registered' + 'casual' is the "cnt" so using these might overfit the model.

```
# casual is not predictor variable so for now ignored it.
## replacing the outliers with Na values

count_names = ['windspeed','hum']
for i in count_names:
    print (i)
    q75,q25 = np.percentile(day.loc[:,i],[75,25])
    iqr = q75-q25
    min = q25 - (iqr*1.5)
    max = q75 + (iqr*1.5)
    print (min)
    print (max)

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

```
windspeed
-0.012446750000000034
0.38061125
hum
0.20468725
1.0455212500000002
```

```
day.isnull().sum() ### shows missing values, 2 in hum and 13 in windspeed
```

```
instant        0
dteday         0
season         0
yr             0
mnth           0
holiday        0
weekday        0
workingday     0
weathersit     0
temp           0
atemp          0
hum            2
windspeed     13
casual         0
registered     0
cnt            0
dtype: int64
```

After this replace the outliers with Na values and drop them. Dropping them because less in quantity won't affect much in model development .

```
day = day.dropna(axis = 0) ## removing the outliers
```

```
day.isnull().sum()
```

```
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
```

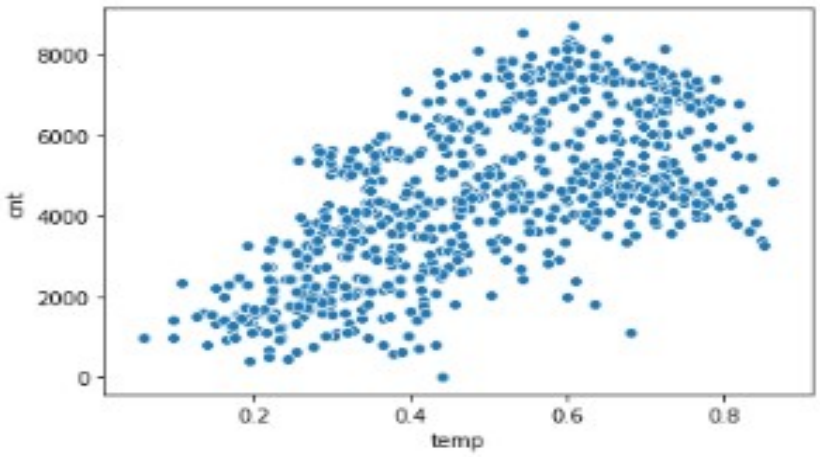
Chapter 4

DATA Visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data. Matplotlib.pyplot and seaborn librabries are used here in python. These scatter pkots and graphs are analyzed to check the relation between independent and target variable

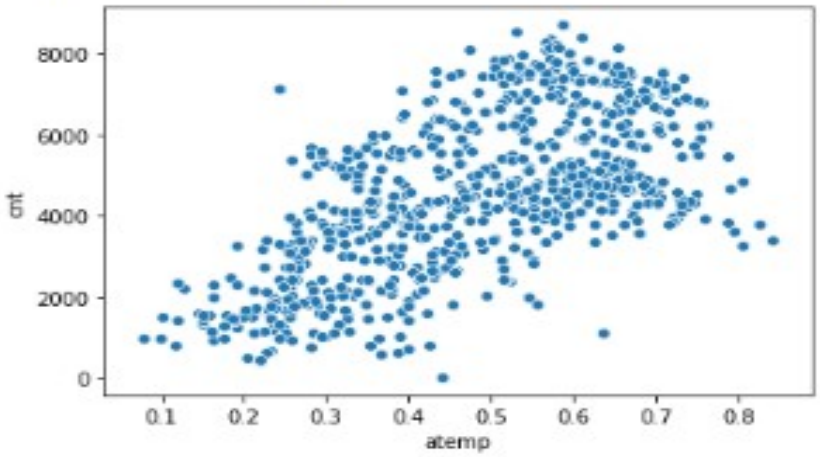
```
### relation between continous variables and target variables
sns.scatterplot(data=day,x='temp',y='cnt')  ### between temp and cnt
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x208173dfa20>
```



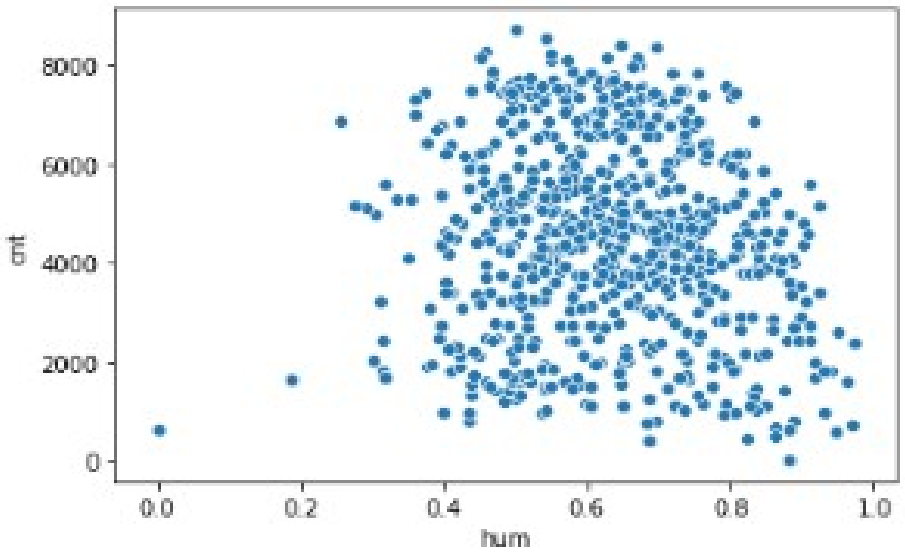

```
sns.scatterplot(data=day,x='atemp',y='cnt')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x208172d7080>
```



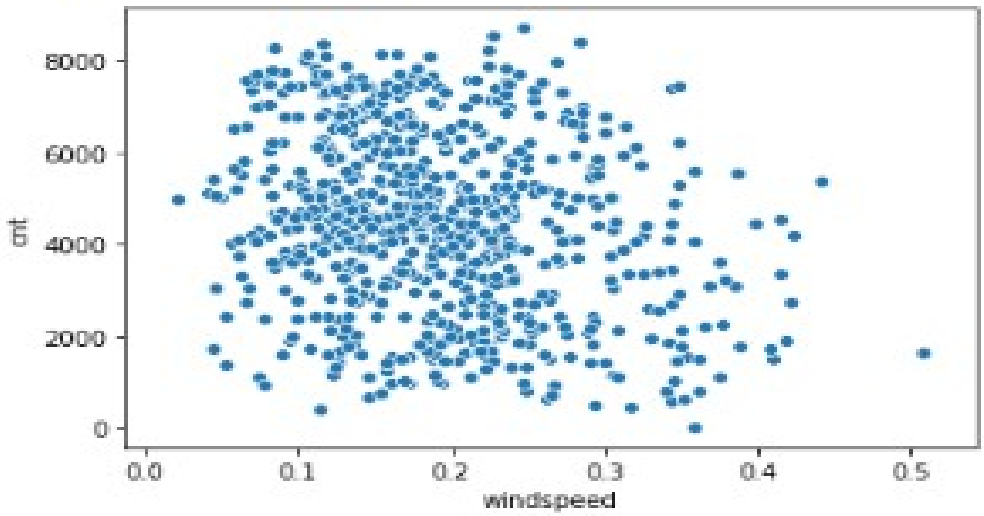
```
sns.scatterplot(data=day,x='hum',y='cnt')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x20817338da0>
```



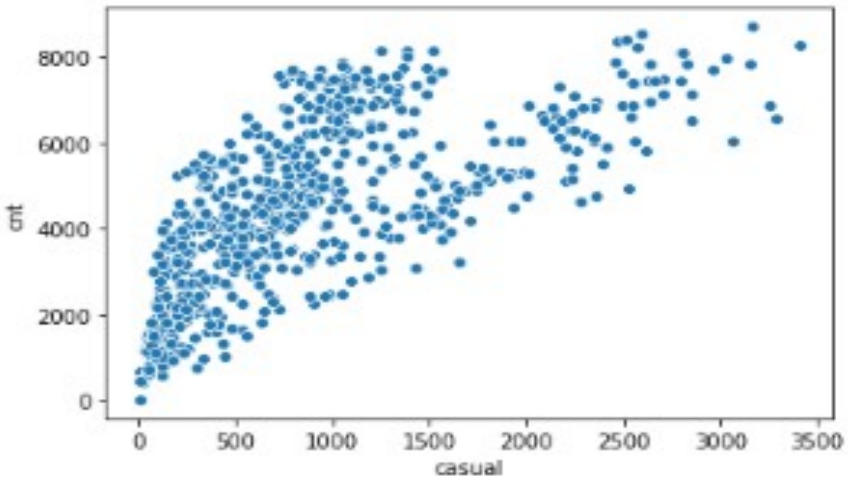

```
sns.scatterplot(data=day,x='windspeed',y='cnt')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2081738fa90>
```



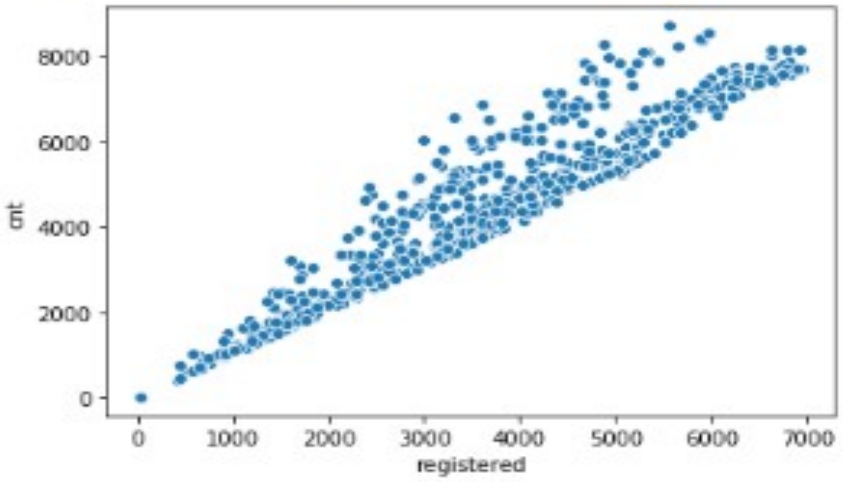
```
sns.scatterplot(data=day,x='casual',y='cnt')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x208174561d0>
```

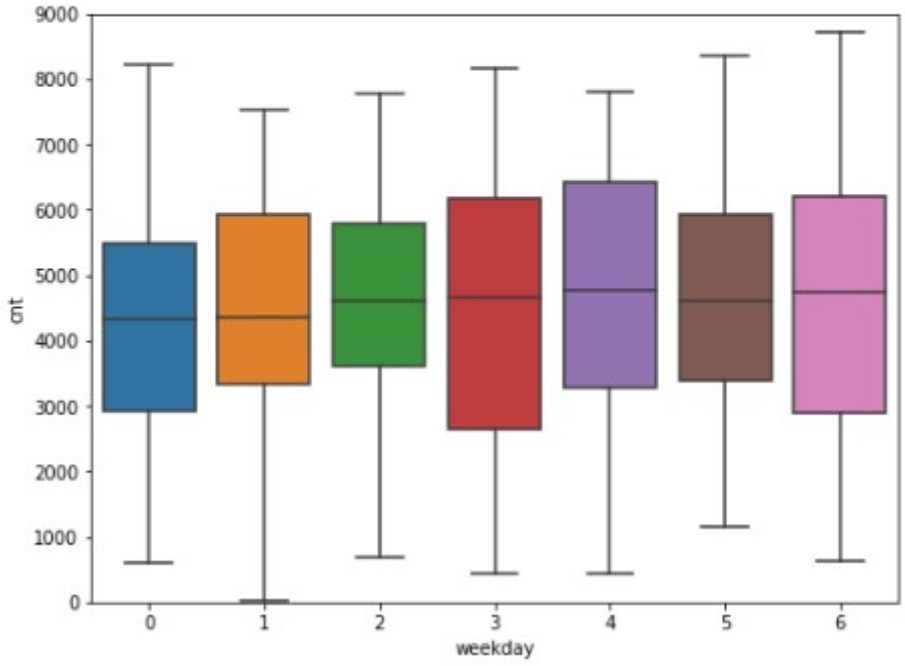



```
sns.scatterplot(data=day,x='registered',y='cnt')
```

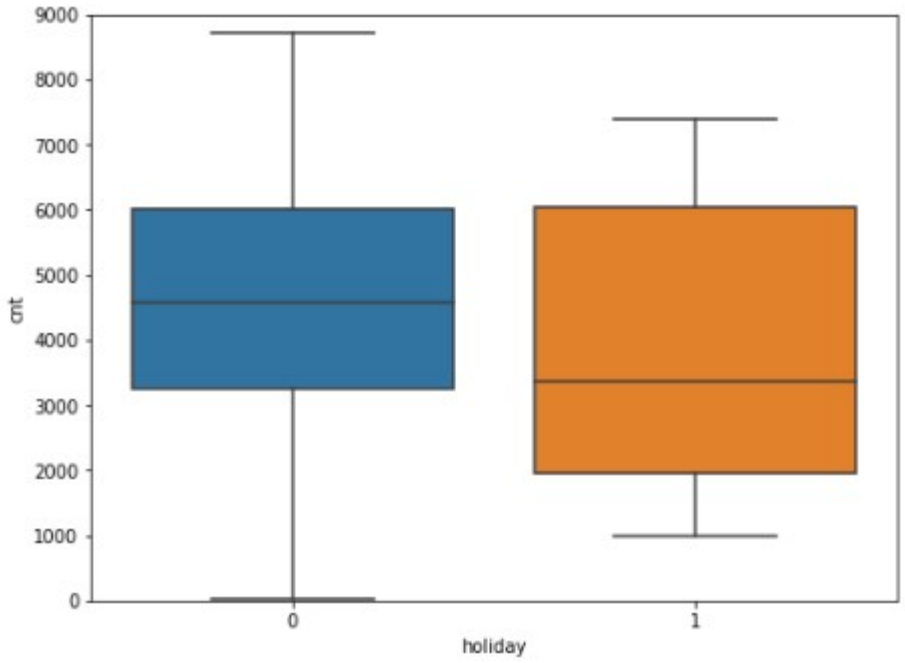
```
<matplotlib.axes._subplots.AxesSubplot at 0x208174b0fd0>
```



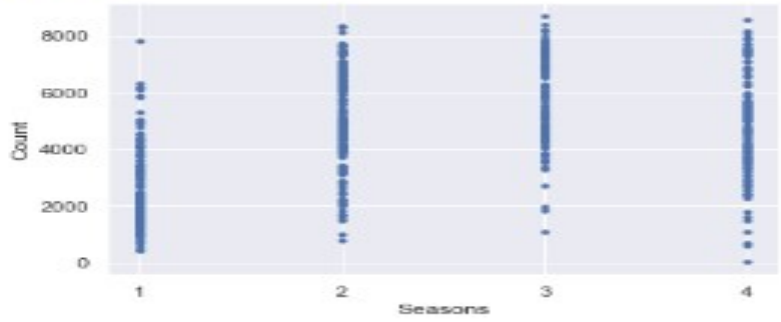
```
## boxplots for categorical variables
weekdays = 'weekday'
data = pd.concat([day['cnt'], day[weekdays]], axis=1)
f, ax = plt.subplots(figsize=(8, 6))
fig = sns.boxplot(x=weekdays, y="cnt", data=data)
fig.axis(ymin=0, ymax=9000);
```




```
holidays = 'holiday'
data = pd.concat([day['cnt'], day[holidays]], axis=1)
f, ax = plt.subplots(figsize=(8, 6))
fig = sns.boxplot(x=holidays, y="cnt", data=data)
fig.axis(ymin=0, ymax=9000);
```

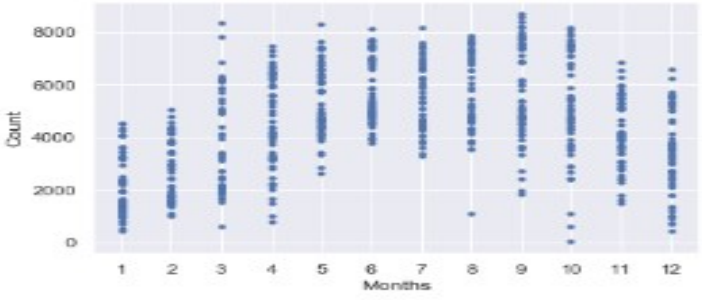


```
# check relationship between count and categorical variables
## season and cnt
plt.scatter(x=day['season'], y=day['cnt'], s=10)
plt.xlabel('Seasons')
plt.ylabel('Count')
plt.show()
```




```
# count and year
plt.scatter(x=day['yr'], y=day['cnt'], s=10)
plt.xlabel('Year')
plt.ylabel('Count')
plt.show()
```


```
# count and months
plt.scatter(x=day['mnth'], y=day['cnt'], s=10)
plt.xlabel('Months')
plt.ylabel('Count')
plt.show()
```




```
# cnt and workingday
plt.scatter(x=day['workingday'], y=day['cnt'], s=10)
plt.xlabel('Working Day')
plt.ylabel('Count')
plt.show()
```

Chapter 5

FEATURE Engineering: feature scaling and feature selection.

For feature selection and scaling, we need to know how the variables are affecting the target variable. What features are important and what might not be that significant. So co relation plot with pearson method is used to check the co relation between the continous variables.

Then vif(variance inflation factor) analysis is performed for numerical variables to check multicolinearity.

And chi^2 test is performed on categorical variables to find out the p value.

```
numeric_day = day.loc[:,['temp','atemp','hum','windspeed','casual','registered','cnt']]
```

```
### co relation plot to check co relation between numerical/continous variables
numeric_day.corr(method='pearson').style.format("{:.2}").background_gradient(cmap=plt.get_cmap('coolwarm'), axis=1)
```

	temp	atemp	hum	windspeed	casual	registered	cnt
temp	1.0	0.99	0.11	-0.14	0.54	0.54	0.63
atemp	0.99	1.0	0.13	-0.17	0.54	0.54	0.63
hum	0.11	0.13	1.0	-0.2	-0.1	-0.12	-0.14
windspeed	-0.14	-0.17	-0.2	1.0	-0.15	-0.2	-0.22
casual	0.54	0.54	-0.1	-0.15	1.0	0.39	0.67
registered	0.54	0.54	-0.12	-0.2	0.39	1.0	0.94
cnt	0.63	0.63	-0.14	-0.22	0.67	0.94	1.0

```
# cheking multicolinerity by VIF(variance inflation factor)

from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant

VIF_day = add_constant(day.iloc[:,9:15])
pd.Series([variance_inflation_factor(VIF_day.values, i)
               for i in range(VIF_day.shape[1])],
              index=VIF_day.columns)
```

```
const         54.847289
temp          63.442490
atemp         64.309759
hum            1.179328
windspeed      1.154450
casual         1.502061
registered     1.561168
dtype: float64
```

```
# chi square test for categorical variables

# saving categorical variables name in categ

categ = []
for i in range(0, day.shape[1]):
    if(day.iloc[:,i].dtypes == 'object'):
        categ.append(day.columns[i])
categ
```

```
['season', 'yr', 'mnth', 'holiday', 'weekday', 'workingday', 'weathersit']
```

```
from scipy.stats import chi2_contingency
## for taking out the p values every combination of categorical variables are used
factors_paired = [(i,j) for i in categ for j in categ]
factors_paired
p_values = []
from scipy.stats import chi2_contingency
for factor in factors_paired:
    if factor[0] != factor[1]:
        chi2, p, dof, ex = chi2_contingency(pd.crosstab(day[factor[0]], day[factor[1]]))
        p_values.append(p.round(3))
    else:
        p_values.append('-')
p_values = np.array(p_values).reshape((7,7))
p_values = pd.DataFrame(p_values, index=categ, columns=categ)
print(p_values)
```

```
            season     yr   mnth holiday weekday workingday weathersit
season           -  0.999    0.0   0.641     1.0      0.946      0.013
yr           0.999      -    1.0   0.995     1.0      0.956      0.183
mnth           0.0    1.0      -   0.571     1.0      0.993       0.01
holiday      0.641  0.995  0.571       -     0.0        0.0      0.599
weekday        1.0    1.0    1.0     0.0       -        0.0      0.249
workingday   0.946  0.956  0.993     0.0     0.0          -      0.294
weathersit   0.013  0.183   0.01   0.599   0.249      0.294          -
```

5.1 Outcomes:-

- heat map and vif suggests atemp is highly corelated to temp
- chi^2 test suggests to remove weekday and holiday as the do not contribute much.
- casual and registered are removed as they sum up to count.
- instant is index and dteday is date which is not useful for building regression model.

After this removed (‘atemp’,’weekday’’holiday’,’casual’,’registered’,’instant’,’dteday’) variables,this is what the cleaned data contains and looks like.

```
day = day.drop(columns=['instant','dteday','atemp','casual','registered','holiday','weekday'])
```

```
day.head()
```

	season	yr	mnth	workingday	weathersit	temp	hum	windspeed	cnt
0	1	0	1	0	2	0.344167	0.805833	0.160446	985
1	1	0	1	0	2	0.363478	0.696087	0.248539	801
2	1	0	1	1	1	0.196364	0.437273	0.248309	1349
3	1	0	1	1	1	0.200000	0.590435	0.160296	1562
4	1	0	1	1	1	0.226957	0.436957	0.186900	1600

Chapter 6

MODEL Development:- the selected variables are now split in test and train dataset for training and evaluation under different machine learning models and their accuracy is calculated and compared for best model to fit on the data for accurate prediction. Here linear regression, decision tree, random forest, and gradient boosting algorithns are used and then after hyperparameter tuning is done on random forest and gradient boosting as the gave promising results, hyperparameter tuning is done through randomisedCV(cross validation) and GridsearchCV using k fold strata.

6.1 Splitting the test and train data and model generation.

```
## splitting the data
from sklearn.model_selection import train_test_split
train, test = train_test_split(day, test_size=0.2)
```

```
train.head()
```

	season	yr	mnth	workingday	weathersit	temp	hum	windspeed	cnt
306	4	0	11	1	1	0.408333	0.702083	0.136817	3974
704	4	1	12	1	1	0.438333	0.485000	0.324021	5729
569	3	1	7	1	1	0.741667	0.694167	0.138683	6066
300	4	0	10	1	2	0.330833	0.585833	0.229479	3747
116	2	0	4	1	2	0.620000	0.835417	0.312200	3872

```
### MODEL generation
train_ft_1 = train[['season','yr','mnth','weathersit','temp','hum','windspeed']].values
train_tgt = train['cnt'].values
test_ft = test[['season','yr','mnth','weathersit','temp','hum','windspeed']].values
test_tgt= test['cnt'].values
train_ft_1
```

```
array([['4', '0', '11', ..., 0.408333, 0.702083, 0.136817],
       ['4', '1', '12', ..., 0.438333, 0.485, 0.324021],
       ['3', '1', '7', ..., 0.741667, 0.694167, 0.138683],
       ...,
       ['3', '1', '7', ..., 0.720833, 0.6675, 0.15173699999999998],
       ['2', '1', '5', ..., 0.6, 0.45625, 0.083975],
       ['2', '1', '3', ..., 0.484167, 0.48125, 0.291671]], dtype=object)
```

6.2 Linear regression model:

Linear regression models are used to show or predict the relationship between two variables or factors. The factor that is being predicted (the factor that the equation *solves for*) is called the dependent variable. The factors that are used to predict the value of the dependent variable are called the independent variables.In linear regression, each observation consists of two values. One value is for the dependent variable and one value is for the independent variable. In this simple model, a straight line approximates the relationship between the dependent variable and the independent variable.When two or more independent variables are used in regression analysis, the model is no longer a simple linear one. This is known as multiple regression.

```
LR = LinearRegression().fit(train_ft_1 ,train_tgt)
```

```
pred_train_LR = LR.predict(train_ft_1)
pred_test_LR = LR.predict(test_ft)
```

```
RMSE_train_LR= np.sqrt(mean_squared_error(train_tgt, pred_train_LR))

# calculate RMSE on test data
RMSE_test_LR = np.sqrt(mean_squared_error(test_tgt, pred_test_LR))
print("training data rmse = "+str(RMSE_train_LR))
print("test data rmse= "+str(RMSE_test_LR))
```

```
training data rmse = 910.2095615984606
test data rmse= 802.0889000812862
```

```
# calculate R^2 on train data
train_LR = r2_score(train_tgt, pred_train_LR)
# calculate R^2 on test data
test_LR = r2_score(test_tgt, pred_test_LR)

print("training r2 = "+str(train_LR))
print("test r2 = "+str(test_LR))
```

```
training r2 = 0.7895782154586652
test r2 = 0.7755791182043209
```

```
errors = abs(pred_test_LR - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
MAPE = 17.51
Accuracy = 82.49%.
```

6.3 Decision tree model:

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

```
### decision tree model
from sklearn.tree import DecisionTreeRegressor
D_tree = DecisionTreeRegressor()
D_tree = D_tree.fit(train_ft_1, train_tgt)
print(D_tree)
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      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,
                      presort=False, random_state=None, splitter='best')
```

```
predict_DT = D_tree.predict(test_ft)
print(predict_DT)
```

```
[3623. 5046.  795. 5923. 4744. 6153. 4338. 3129. 4308. 8090. 6855. 8555.
 7384. 4274. 7582. 4790. 3974. 2765. 4634. 2177. 3542. 6824. 5668. 5992.
 7733. 1526. 5668. 3894. 7384. 1708. 6536.  605. 4585.   22. 5062. 3129.
 6597. 4629. 4075. 2424. 4648. 4911. 4189. 3351. 7375. 4266. 3956. 7852.
 6883. 1834. 2743. 6998. 7261. 5204. 4120. 4744. 2425. 4985. 1096. 4906.
 4763. 5202. 6786. 3956. 7129. 2703. 4367. 5687. 4105. 7436. 4150.  822.
 7148. 5611. 4010. 5041. 4352. 2999. 6569. 6565. 4586. 1471. 5362. 6772.
 3907. 4911. 1996. 4075. 2918. 5260. 5923. 4086. 4862. 3614. 8555. 5342.
 4120. 3071. 3922. 7572. 2133. 4773. 7359. 1321. 4773. 6784. 5870. 4727.
 4717. 5035. 8294. 4401. 4548. 6857. 8120. 3392. 5633. 4760. 4367. 4352.
 5260. 6421. 7605. 3840. 4186. 6883. 5992. 5532. 2594. 3163. 3126. 6591.
 3747. 3542. 3598. 2311. 4098. 1471. 5976. 2843. 3958. 3956. 3907. 7105.]
```

```
def MAPE(y_true, y_pred):
    mape = np.mean(np.abs((y_true - y_pred) / y_true))*100
    return mape
MAPE(test_tgt, predict_DT)
```

```
16.70518821693581
```

```
errors = abs(predict_DT - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
MAPE = 16.71
Accuracy = 83.29%.
```

6.4 Random forest model:

The **random forest** is a classification algorithm consisting of many decisions trees. It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated **forest** of trees whose prediction by committee is more accurate than that of any individual tree.

```
### random forest model
from sklearn.ensemble import RandomForestRegressor
```

```
RF_model_one = RandomForestRegressor(n_estimators= 500, random_state=100).fit(train_ft_1,train_tgt)
RF_predict_one= RF_model_one.predict(test_ft)
```

```
def RMSE(y_test,y_predict):
    mse = np.mean(y_test-y_predict)
    print("Mean Square : ",mse)
    rmse=np.sqrt(mse)
    print("Root Mean Square : ",rmse)
    return rmse
```

```
RMSE(test_tgt,RF_predict_one)
```

```
Mean Square :  3.0912777777777922
Root Mean Square :  1.7582029967491786

1.7582029967491786
```

```
MAPE(test_tgt,RF_predict_one)
```

```
14.374174158646333
```

```
errors = abs(RF_predict_one - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
MAPE = 14.37
Accuracy = 85.63%.
```

6.5 Gradient Boosting Model:

Gradient boosting is a type of machine learning boosting. It relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error. How are the targets calculated? The target outcome for each case in the data depends on how much changing that case's prediction impacts the overall prediction error

```
# GRADIENT BOOSTING MODEL
GB = GradientBoostingRegressor().fit(train_ft_1, train_tgt)
# predict on train data
train_GB = GB.predict(train_ft_1)

# predict on test data
test_GB = GB.predict(test_ft)
```

```
# RMSE on train data
RMSE_train_GB = np.sqrt(mean_squared_error(train_tgt, train_GB))
# RMSE on test data
RMSE_test_GB = np.sqrt(mean_squared_error(test_tgt, test_GB))
print("training rmse = "+str(RMSE_train_GB))
print("test rmse = "+str(RMSE_test_GB))
```

```
training rmse = 429.1444865932088
test rmse = 684.0966697404792
```

```
# R^2 of train data
r2_train_GB = r2_score(train_tgt, train_GB)
# R^2 of test data
r2_test_GB = r2_score(test_tgt, test_GB)
```

```
print("r^2 train data = "+str(r2_train_GB))
print("r^2 test data = "+str(r2_test_GB))
```

```
r^2 train data = 0.9532248717091775
r^2 test data = 0.8367499945775301
```

```
errors = abs(test_GB - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
MAPE = 14.65
Accuracy = 85.35%.
```

6.6 Hyper parameter tuning using:

6.6.1 RandomisedsearchCV on Random forest

efficient way to find an optimal set of hyperparameters for a machine learning model is to use random search. The randomized search meta-estimator is an algorithm that trains and evaluates a series of models by taking random draws from a predetermined set of hyperparameter distributions. The algorithm picks the most successful version of the model it's seen after training N different versions of the model with different randomly selected hyperparameter combinations, leaving you with a model trained on a near-optimal set of hyperparameters.

```
from sklearn.model_selection import RandomizedSearchCV
```

```
### hyperparameter tuning on Randomforest using randomisedSearchCV
RFR = RandomForestRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator, 'max_depth': depth}

RCV_rf = RandomizedSearchCV(RFR, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
RCV_rf = RCV_rf.fit(train_ft_1, train_tgt)
predictions_RFR = RCV_rf.predict(test_ft)

best_params_RFR = RCV_rf.best_params_

best_estimator_RFR = RCV_rf.best_estimator_

predictions_RFR = best_estimator_RFR.predict(test_ft)

# calculate R^2
RFR_r2 = r2_score(test_tgt, predictions_RFR)

# calculate RMSE
RFR_rmse = np.sqrt(mean_squared_error(test_tgt, predictions_RFR))

# calculate mape and accuracy
errors = abs(predictions_RFR - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape

print('RandomizedSearchCV - Random Forest Regressor Model Performance:')
print('Best Parameters = ',best_params_RFR)
print('R-squared = {:0.2}.'.format(RFR_r2))
print('RMSE = ',RFR_rmse)
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
RandomizedSearchCV - Random Forest Regressor Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.83.
RMSE =  688.201574846633
MAPE = 14.46
Accuracy = 85.54%.
```

6.6.2 RandomsearchCV on gradient boosting model

```
# Gradientboosting model hyperparameter tuning by randomisedsearchCV

GBR = GradientBoostingRegressor(random_state = 0)
n_estimator = list(range(1,20,2))
depth = list(range(1,100,2))

# Create the random grid
rand_grid = {'n_estimators': n_estimator, 'max_depth': depth}

RCV_GB = RandomizedSearchCV(GBR, param_distributions = rand_grid, n_iter = 5, cv = 5, random_state=0)
RCV_GB = RCV_GB.fit(train_ft_1, train_tgt)
predictions_gb = RCV_GB.predict(test_ft)

best_params_gb = RCV_GB.best_params_

best_estimator_gb = RCV_GB.best_estimator_

predictions_gb = best_estimator_gb.predict(test_ft)

# calculate R^2
GB_r2 = r2_score(test_tgt, predictions_gb)

# calculate RMSE
GB_rmse = np.sqrt(mean_squared_error(test_tgt, predictions_gb))

# calculate mape and accuracy
errors = abs(predictions_gb - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape

print('RandomizedSearchCV - Gradient Boosting Model Performance:')
print('Best Parameters = ',best_params_gb)
print('R-squared = {:0.2}.'.format(GB_r2))
print('RMSE = ', GB_rmse)
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
RandomizedSearchCV - Gradient Boosting Model Performance:
Best Parameters =  {'n_estimators': 15, 'max_depth': 9}
R-squared = 0.77.
RMSE =  810.7275156370196
MAPE = 17.34
Accuracy = 82.66%.
```

6.6.3 GridsearchCV on random forest model

Grid search is the process of performing hyper parameter tuning in order to determine the optimal values for a given model. This is significant as the performance of the entire model is based on the hyper parameter values specified.

```
###Hyperparameter tuning using GridSearchCV

from sklearn.model_selection import GridSearchCV

### 1. GridSearchCV on Random Forest Model

rfr_gs = RandomForestRegressor(random_state = 0)
n_estimator = list(range(11,20,1))
depth = list(range(5,15,2))

# Create the grid
grid_search = {'n_estimators': n_estimator, 'max_depth': depth}

## Grid Search Cross-Validation with 5 fold CV
gridcv_rf = GridSearchCV(rfr_gs, param_grid = grid_search, cv = 5)
gridcv_rf = gridcv_rf.fit(train_ft_1,train_tgt)

best_params_GRF = gridcv_rf.best_params_
best_estimator_GRF = gridcv_rf.best_estimator_

#Apply model on test data
predictions_GRF = best_estimator_GRF.predict(test_ft)

# calculate R^2
GRF_r2 = r2_score(test_tgt, predictions_GRF)

# calculate RMSE
GRF_rmse = np.sqrt(mean_squared_error(test_tgt, predictions_GRF))

# calculate mape and accuracy
errors = abs(predictions_GRF - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape

print('GridSearchCV - Random Forest Regressor Model Performance:')
print('Best Parameters = ',best_params_GRF)
print('R-squared = {:0.2}.'.format(GRF_r2))
print('RMSE = ',(GRF_rmse))
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
GridSearchCV - Random Forest Regressor Model Performance:
Best Parameters =  {'max_depth': 9, 'n_estimators': 17}
R-squared = 0.83.
RMSE =  701.1655257383472
MAPE = 14.71
Accuracy = 85.29%.
```

6.6.4 GridsearchCV on Gradient boosting model

```
GridSearchCV on gradient boosting model

gbr_gs = GradientBoostingRegressor(random_state = 0)
n_estimator = list(range(11,20,1))
depth = list(range(5,15,2))

# Create the grid
grid_search = {'n_estimators': n_estimator, 'max_depth': depth}

# Grid Search Cross-Validation with 5 fold CV
gridcv_gb = GridSearchCV(gbr_gs, param_grid = grid_search, cv = 5)
gridcv_gb = gridcv_gb.fit(train_ft_1,train_tgt)

best_params_Ggb = gridcv_gb.best_params_
best_estimator_Ggb = gridcv_gb.best_estimator_

#Apply model on test data
predictions_Ggb = best_estimator_Ggb.predict(test_ft)

# calculate R^2
Ggb_r2 = r2_score(test_tgt, predictions_Ggb)

# calculate RMSE
Ggb_rmse = np.sqrt(mean_squared_error(test_tgt, predictions_Ggb))

# calculate mape and accuracy
errors = abs(predictions_Ggb - test_tgt)
mape = 100 * np.mean(errors / test_tgt)
accuracy = 100 - mape

print('Grid Search CV Gradient Boosting regression Model Performance:')
print('Best Parameters = ',best_params_Ggb)
print('R-squared = {:0.2}.'.format(Ggb_r2))
print('RMSE = ',(Ggb_rmse))
print('MAPE = {:0.2f}'.format(mape))
print('Accuracy = {:0.2f}%.'.format(accuracy))
```

```
Grid Search CV Gradient Boosting regression Model Performance:
Best Parameters =  {'max_depth': 5, 'n_estimators': 19}
R-squared = 0.81.
RMSE =  733.2511004937871
MAPE = 16.03
Accuracy = 83.97%.
```

Chapter 7

RESULTS AND CONCLUSION

MODEL	MAPE	RMSE	ACCURACY
Linear regression	17.51	802.08	82.49%
Decision tree	16.71	2.50	83.29%
Decision tree 2	15.95	6.411	84.05%
Random forest	14.37	1.758	85.63%
Gradient Boosting	14.65	684.096	85.35%
RCV_Random forest	14.48	688.20	85.54%
RCV_Gradient boosting	17.34	810.72	82.66%
GCV_Random forest	14.71	701.65	85.29%
GCV_Gradient boosting	16.03	733.25	83.97%

From the results it concludes that all the models work well, while the random forest model is termed to be more accurate and can be used for better results. Also randomized search CV on random forest gives nearly similar values to random forest so this cross validation technique works better than grid CV for this data.

Similar procedure have been followed in R program file there also similar results were found. So I conclude that random forest model works well on the given dataset.

Appendix: R code

```
rm(list=ls())

setwd("C:/Users/17519")## setting the working directory

getwd()


# loading Libraries

x = c("rpart", "randomForest","xgboost","caret", "DMwR","tidyr", "ggplot2", "corrgram", "usdm")


# rpart - decision tree

# randomForest - random forest

# xgboost - xgboost

# caret - createDataPartition

# DMwR - regr.eval

# tidyr - drop_na

# ggplot2 - for visulization, boxplot, scatterplot

# corrgram - correlation plot

# usdm - vif

# loading Packages by lapply function

lapply(x, require, character.only = TRUE)

rm(x)

day = read.csv("C:/Users/17519/day.csv",header=T)## importing the dataset

head(day)
```

```
summary(day)


## data preprocessing and exploration


# data types arre changed as per required
day$dteday = as.Date(as.character(day$dteday))


## all the categorical variables as factors
cnames=c("season","yr","mnth","holiday","weekday","workingday","weat
hersit")
for(i in cnames){
  print(i)
  day[,i]=as.factor(day[,i])
}
# Missing values in data
apply(day, 2, function(x) {sum(is.na(x))})


## defining a variable to store numerical variables
num_index = sapply(day, is.numeric)
numeric_data = day[,num_index]
num_cnames = colnames(numeric_data)


## outlier analysis boxplot is used


  ggplot(data = day, aes(x = "", y = casual)) +
```

```
geom_boxplot()
ggplot(data = day, aes(x = "", y = temp)) +
  geom_boxplot()
ggplot(data = day, aes(x = "", y = atemp)) +
  geom_boxplot()
ggplot(data = day, aes(x = "", y = hum)) +
  geom_boxplot()
ggplot(data = day, aes(x = "", y = windspeed)) +
  geom_boxplot()
ggplot(data = day, aes(x = "", y = registered)) +
  geom_boxplot()
outlier_var=c("hum","windspeed")### outliers are observed in these 2

#Replace all outliers with NA
for(i in outlier_var){
  val = day[,i][day[,i] %in% boxplot.stats(day[,i])$out]
  print(length(val))
  day[,i][day[,i] %in% val] = NA
}
apply(day, 2, function(x) {sum(is.na(x))})

day = drop_na(day)

df_new = day
```

```
## relation between independent variables and target variables using scatter plot
# Scatter plot between temp and cnt
ggplot(data = day, aes_string(x = day$temp, y = day$cnt))+
  geom_point()
# Scatter plot between atemp and cnt
ggplot(data = day, aes_string(x = day$atemp, y = day$cnt))+
  geom_point()

# Scatter plot between hum and cnt
ggplot(data = day, aes_string(x = day$hum, y = day$cnt))+
  geom_point()

# Scatter plot between windspeed and cnt
ggplot(data = day, aes_string(x = day$windspeed, y = day$cnt))+
  geom_point()

# Scatter plot between season and cnt
ggplot(data = day, aes_string(x = day$season, y = day$cnt))+
  geom_point()

# Scatter plot between month and cnt
ggplot(data = day, aes_string(x = day$mnth, y = day$cnt))+
  geom_point()
```

```
# Scatter plot between weekday and cnt
ggplot(data = day, aes_string(x = day$weekday, y = day$cnt))+
  geom_point()

ggplot(data = day, aes_string(x = day$holiday, y = day$cnt))+
  geom_point()

### features selection
## correlation plot between numeric variables

numeric_index=sapply(day, is.numeric)
corrgram(day[,numeric_index], order=F, upper.panel=panel.pie,
         text.panel=panel.txt, main="Correlation plot")
# check VIF
vif(day[,10:15])
# if vif is greater than 10 then variable is not suitable/multicollinerity

# ANOVA test for checking p-values of categorical variables
for (i in cnames) {
  print(i)
  print(summary(aov(day$cnt ~day[,i], day)))
}
#heat map and vif suggests atemp is highly corelated to temp
```

```
#chi^2 test suggests to remove weekday and holiday as the do not contribute much.

#casual and registered are removed as they sum up to count.

#instant is index and dteday is date which is not useful for building regression model.


# remove the variables

day=subset(day,select=-c(instant,dteday,atemp,casual,registered,holiday,weekday))


df1 = day


#### splitting the data into train and test

set.seed(1234)

library(caret)

train.index = createDataPartition(day$cnt, p = .80, list = FALSE)

train = day[ train.index,]

test  = day[-train.index,]


#### LINEAR REGRESSION MODEL####

LR_model = lm(cnt ~., data=train)

# summary of trained model

summary(LR_model)


LR_prediction = predict(LR_model,test[,1:8])## predicting on test data
```

```
regr.eval(test[,9],LR_prediction)### regression metrics for evaluation

### Decision tree model###
Dtree = rpart(cnt ~ ., data=train, method = "anova")

# summary on trainned model
summary(Dtree)

#Prediction on test_data
prediction_DT = predict(Dtree, test[,1:8])

regr.eval(test[,9], prediction_DT)



#############  Random forest Model ######################
rforest = randomForest(cnt ~., data=train)

# summary on trained model
summary(rforest)

# prediction of test_data
rf_prediction = predict(rforest, test[,1:8])
```

```
regr.eval(test[,9], rf_prediction)



############  XGBOOST Model ##############################

train_matrix = as.matrix(sapply(train[-9],as.numeric))

test_matrix = as.matrix(sapply(test[-9],as.numeric))


xgboost_model = xgboost(data = train_matrix,label = train$cnt, nrounds = 15,verbose = FALSE)


# summary of trained model

summary(xgboost_model)


# prediction on test_data

xgb_prediction = predict(xgboost_model,test_matrix)


regr.eval(test[,9], xgb_prediction)
```

here xgboost and random forest are nearly close so any of the models can be used xgboost is bit more accurate and has better error metric values while the random forest has bit less than that of xgboost model, henceforth they are nearly same in results.