Advanced Machine Learning - Russo Federica

June 12, 2021

Table of Contents

1 Metro Interstate Traffic Volume Prediction

1.1 Introduction

The dataset, which is collected from 2012-2018, describe the Hourly Interstate 94 Westbound traffic volume for MN DoT ATR station 301, roughly midway between Minneapolis and St Paul, MN

Traffic Data were collected by Minnesota Department of Transportation, with an Automatic Traffic Recorder (ATR) which is a permanent device in the pavement that automatically and continuously collects traffic data.

Weather Data were collected by OpenWeatherMap, where provides radar-based nowcasts, weather satellite data and the vast network of weather stations, rain gauges and other weather sensors.

The dataset is composed by 48204 rows and 9 colums:

- *holiday* (Categorical): US National holidays plus regional holiday, Minnesota State Fair
- *temp* (Numeric): Average temp in kelvin
- *rain_1h* (Numeric): Amount in mm of rain that occurred in the hour
- *snow_1h* (Numeric): Amount in mm of snow that occurred in the hour
- *clouds_all* (Numeric): Percentage of cloud cover
- *weather_main* (Categorical): Short textual description of the current weather
- *weather_description* (Categorical): Longer textual description of the current weather
- *date_time* (DateTime): Hour of the data collected in local CST time
- ***traffic_volume*** (Numeric): Hourly I-94 ATR 301 reported westbound traffic volume (the target)

The **aim of the work** is to build a model to predict traffic volume for a specific point of date and time, given the climatic conditions like rainfall, temperature, percentage of the cloud cover, snowfall and the textual description of climate.

[1]:
```python
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

[2]:
```python
metro_data = pd.read_csv('C:/Users/Federica/Desktop/Fede Unict/Advanced Machine␣
↪Learning/Esame/Metro Interstate/Metro_Interstate_Traffic_Volume.csv')
```

[3]:
```python
metro_data
```

[3]:
```
      holiday    temp  rain_1h  snow_1h  clouds_all  weather_main  \
0        None  288.28      0.0      0.0          40        Clouds
1        None  289.36      0.0      0.0          75        Clouds
2        None  289.58      0.0      0.0          90        Clouds
3        None  290.13      0.0      0.0          90        Clouds
4        None  291.14      0.0      0.0          75        Clouds
…         …       …        …        …           …
48199    None  283.45      0.0      0.0          75        Clouds
48200    None  282.76      0.0      0.0          90        Clouds
48201    None  282.73      0.0      0.0          90  Thunderstorm
48202    None  282.09      0.0      0.0          90        Clouds
48203    None  282.12      0.0      0.0          90        Clouds

          weather_description             date_time  traffic_volume
0            scattered clouds   2012-10-02 09:00:00            5545
1               broken clouds   2012-10-02 10:00:00            4516
2             overcast clouds   2012-10-02 11:00:00            4767
3             overcast clouds   2012-10-02 12:00:00            5026
```

```
4                broken clouds  2012-10-02 13:00:00            4918
…                        …                   …              …
48199            broken clouds  2018-09-30 19:00:00            3543
48200          overcast clouds  2018-09-30 20:00:00            2781
48201  proximity thunderstorm  2018-09-30 21:00:00            2159
48202          overcast clouds  2018-09-30 22:00:00            1450
48203          overcast clouds  2018-09-30 23:00:00             954

[48204 rows x 9 columns]
```

[4]:
```
metro_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48204 entries, 0 to 48203
Data columns (total 9 columns):
holiday                48204 non-null object
temp                   48204 non-null float64
rain_1h                48204 non-null float64
snow_1h                48204 non-null float64
clouds_all             48204 non-null int64
weather_main           48204 non-null object
weather_description    48204 non-null object
date_time              48204 non-null object
traffic_volume         48204 non-null int64
dtypes: float64(3), int64(2), object(4)
memory usage: 3.3+ MB
```

There are no null values.

2 Exploratory data analysis

2.1 Holiday

[5]:
```
metro_data['holiday'].value_counts()
```

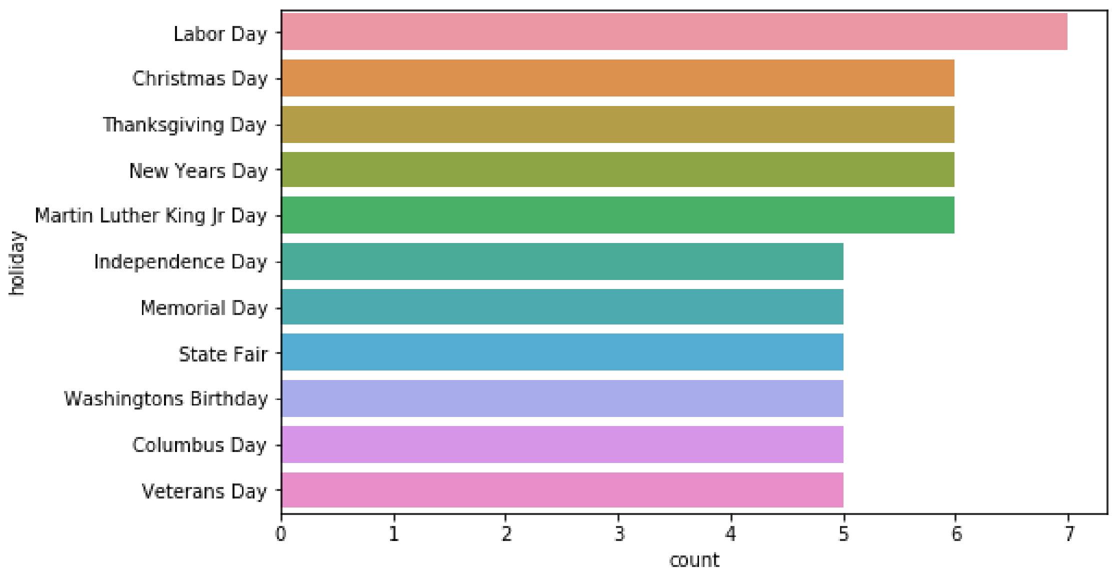
[5]:
```
None                         48143
Labor Day                        7
Thanksgiving Day                 6
New Years Day                    6
Martin Luther King Jr Day        6
Christmas Day                    6
State Fair                       5
Columbus Day                     5
Independence Day                 5
Washingtons Birthday             5
Memorial Day                     5
Veterans Day                     5
Name: holiday, dtype: int64
```

[6]:
```
#The nunique() function in Pandas returns a series with several distinct␣
↪observations in a column.
metro_data['holiday'].nunique()
```

[6]: 12

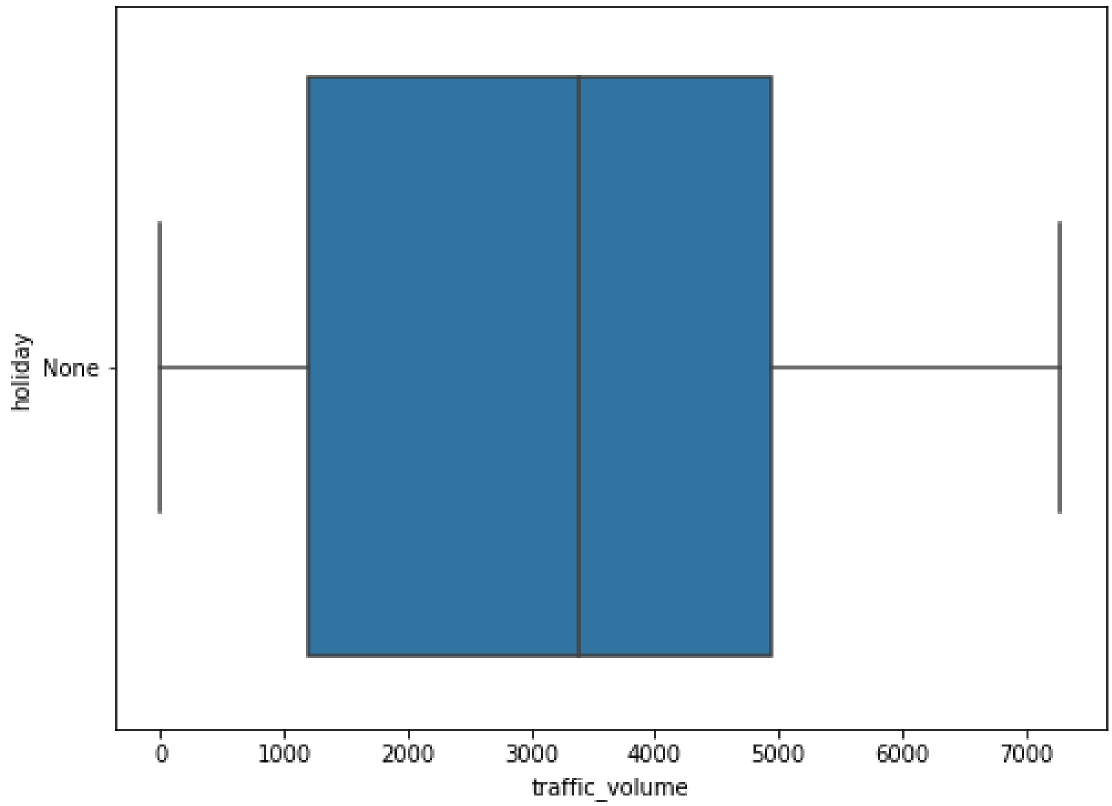
First, I created a bar chart removing the rows 'None' to have a look just of the holidays.

[7]:
```
holidays = metro_data.loc[metro_data.holiday != 'None']
plt.figure(figsize=(8,5))
sns.countplot(y='holiday', data= holidays, order = holidays['holiday'].
↪value_counts().index)
plt.show()
```

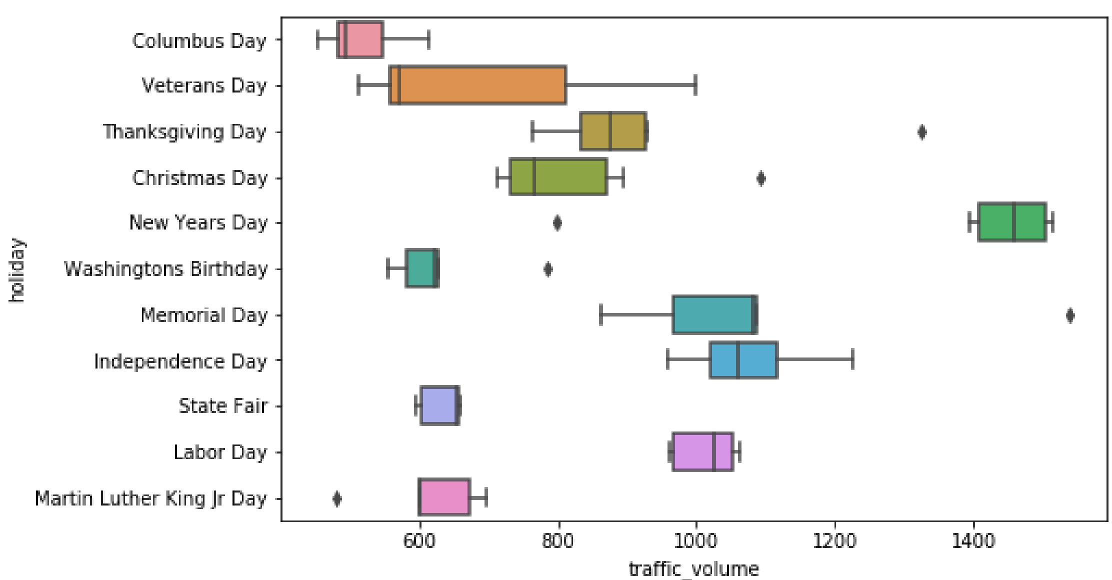


Then, I created conditional box plots of the variable 'no_holidays' and of the variable 'holidays' in combination with our target variable (Traffic_volume).

[8]:
```
no_holidays = metro_data.loc[metro_data.holiday == 'None']
plt.figure(figsize=(8,6))
sns.boxplot(y='holiday',x='traffic_volume', data = no_holidays)
plt.show()
```



[9]:
```
#Exploring traffic volume on holidays
plt.figure(figsize=(8,5))
sns.boxplot(y='holiday',x='traffic_volume', data = holidays)
plt.show()
```



The distribution of the traffic volume during the public holidays has on average low values. There is an exception: “New Years Day” is a holiday that reaches very high traffic volume.

2.2 Temperature

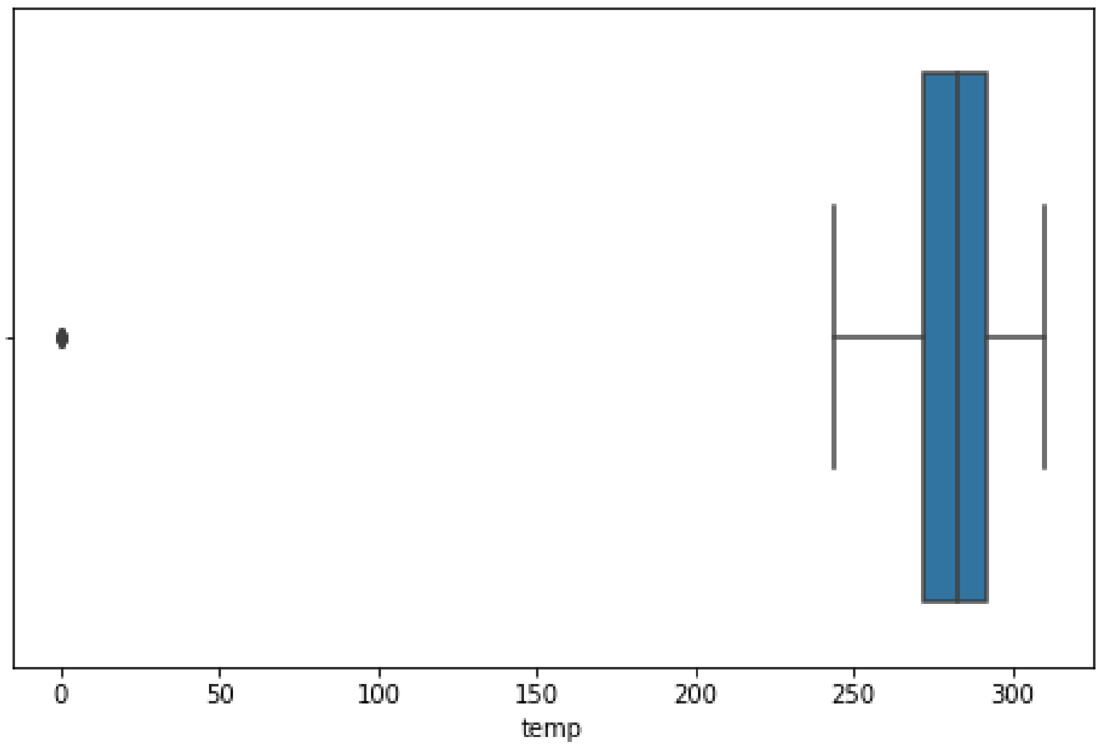
[10]:
```
metro_data['temp'].describe()
```

[10]:
```
count    48204.000000
mean       281.205870
std         13.338232
min          0.000000
25%        272.160000
50%        282.450000
75%        291.806000
max        310.070000
Name: temp, dtype: float64
```

[11]:
```
metro_data['temp'].nunique()
```

[11]:
```
5843
```

[12]:
```
plt.figure(figsize=(8,5))
sns.boxplot('temp', data = metro_data)
plt.show()
```



Temperature feature has an anamoly value around 0. Such observation can be a reading from a faulty sensor. I will eliminate in the data processing phase.

2.3 Rain_1h

[13]:
```
metro_data['rain_1h'].describe()
```

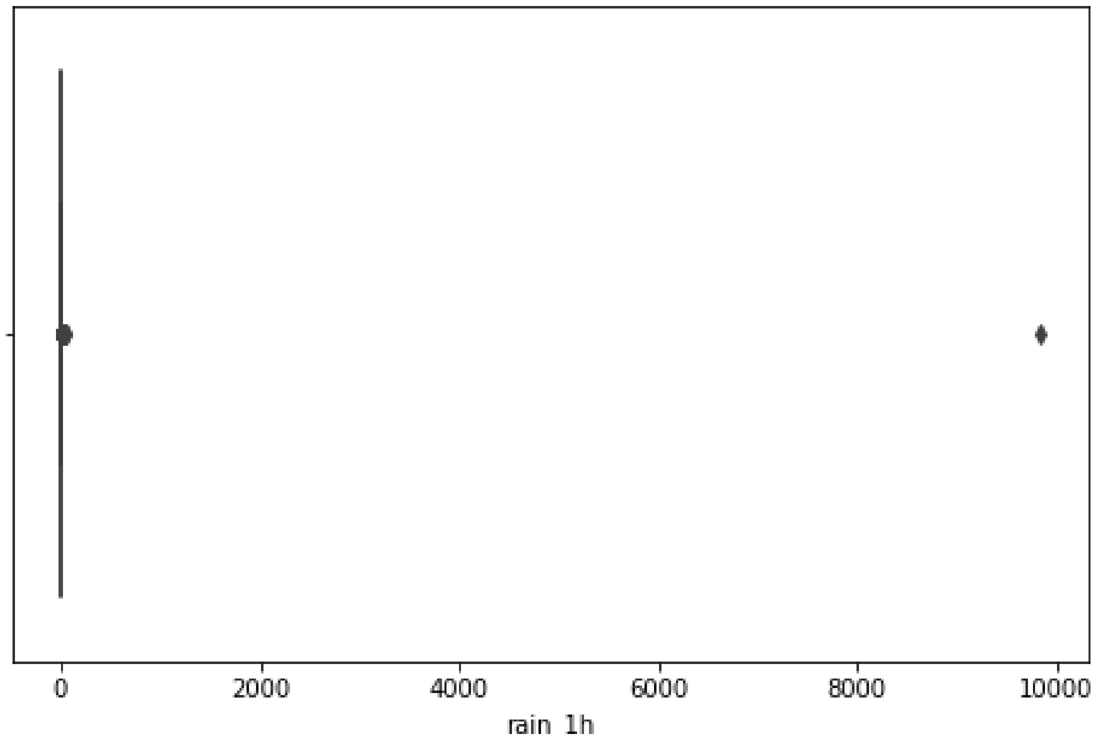
[13]:
```
count    48204.000000
mean         0.334264
std         44.789133
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max       9831.300000
Name: rain_1h, dtype: float64
```

[14]:
```
metro_data['rain_1h'].nunique()
```

[14]:
```
372
```

[15]:
```
plt.figure(figsize=(8,5))
sns.boxplot('rain_1h', data = metro_data)
```

```
plt.show()
```



The boxplot reveals an outlier value approximately at 10000. Also in this case I will go to remove it during the pre-processing.

2.4 Snow_1h

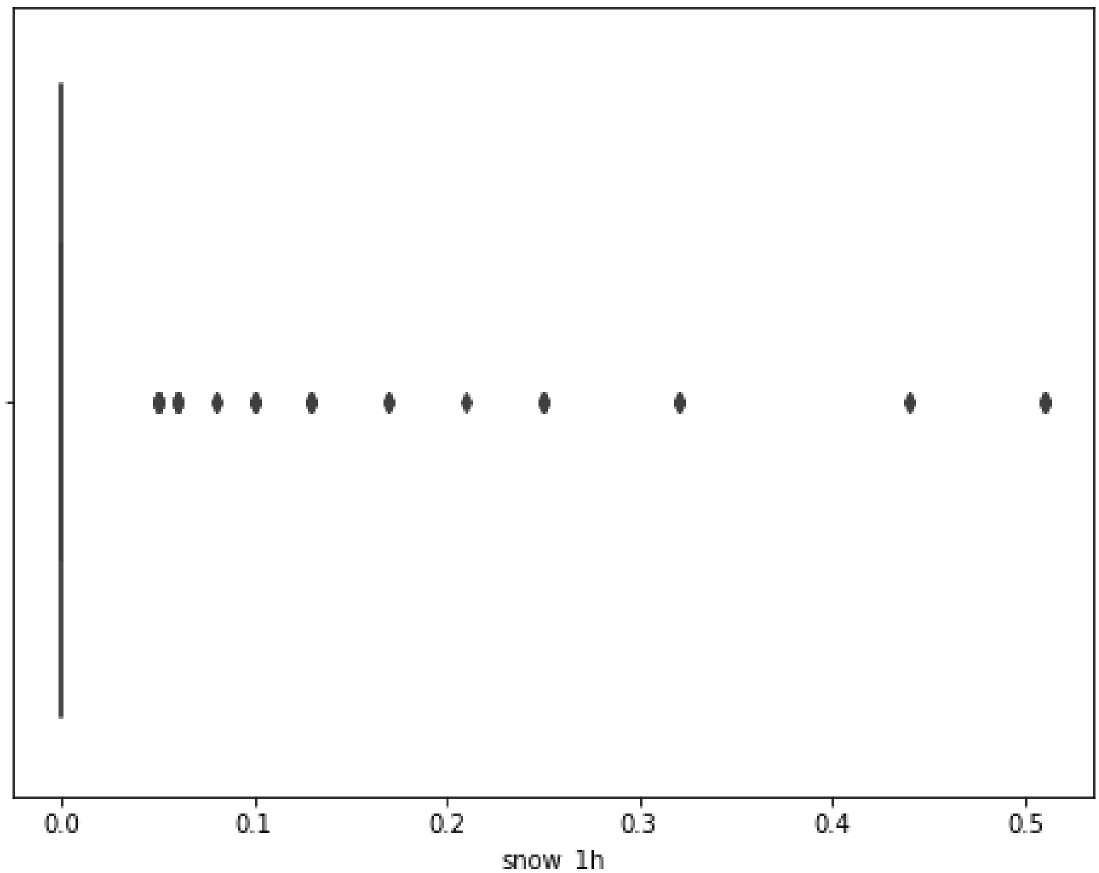
[16]:
```
metro_data['snow_1h'].describe()
```

[16]:
```
count    48204.000000
mean         0.000222
std          0.008168
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max          0.510000
Name: snow_1h, dtype: float64
```

[17]:
```
metro_data['snow_1h'].nunique()
```

[17]:
```
12
```

[18]:
```
plt.figure(figsize=(8,6))
sns.boxplot('snow_1h', data = metro_data)
plt.show()
```



2.5 Clouds_all

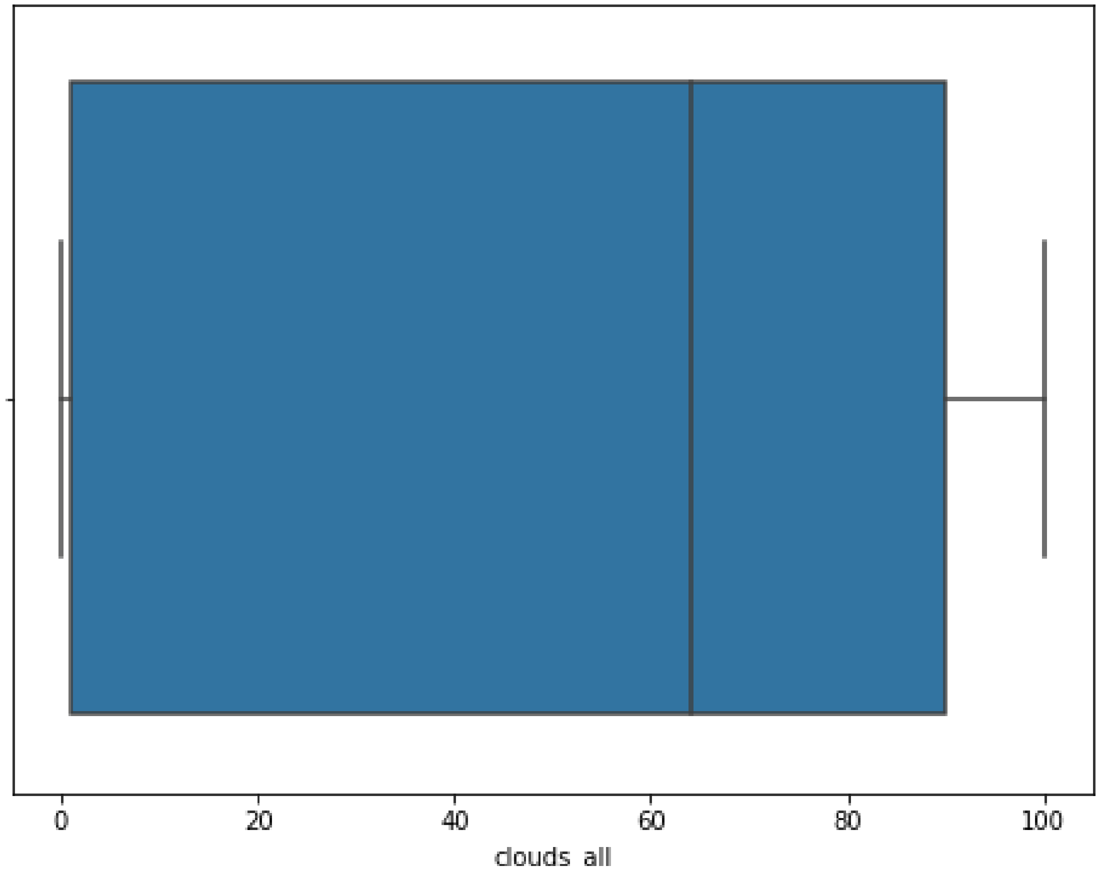
[19]:
```
metro_data['clouds_all'].describe()
```

[19]:
```
count    48204.000000
mean        49.362231
std         39.015750
min          0.000000
25%          1.000000
50%         64.000000
75%         90.000000
max        100.000000
Name: clouds_all, dtype: float64
```

[20]:
```
metro_data['clouds_all'].nunique()
```

[20]: 60

[21]:
```
plt.figure(figsize=(8,6))
sns.boxplot('clouds_all', data = metro_data)
plt.show()
```



2.6 Weather_main

[22]:
```
metro_data['weather_main'].value_counts()
```

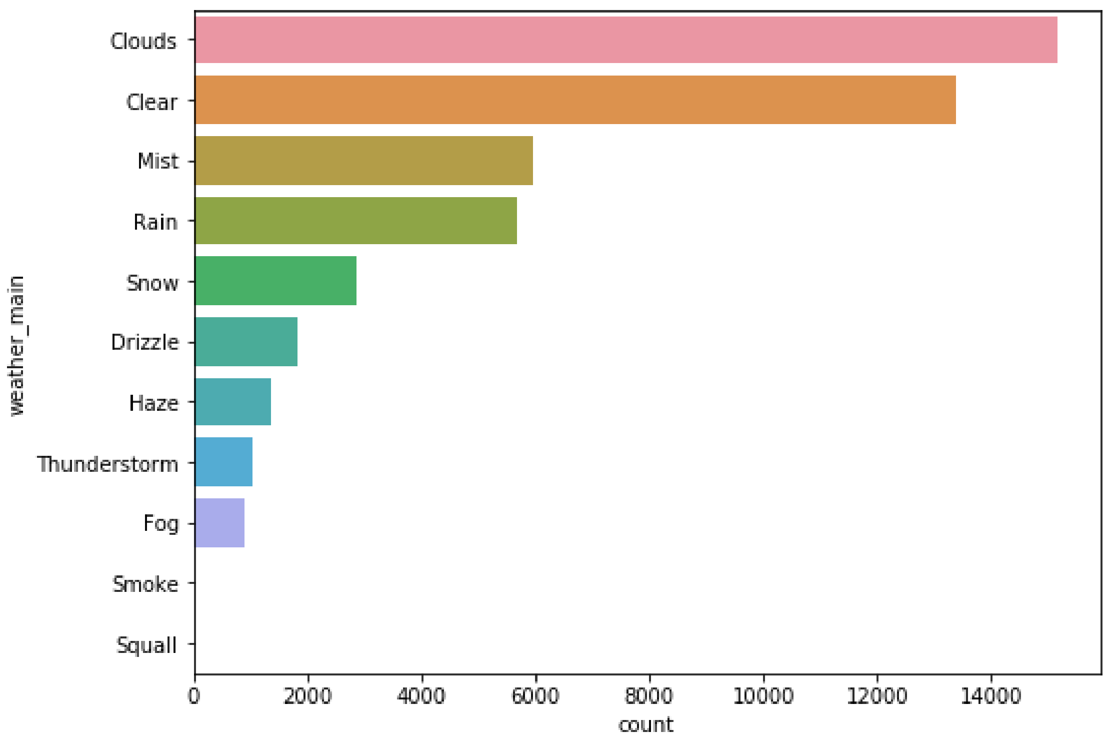
[22]:
```
Clouds          15164
Clear           13391
Mist             5950
Rain             5672
Snow             2876
Drizzle          1821
Haze             1360
Thunderstorm     1034
Fog               912
```

```
Smoke              20
Squall              4
Name: weather_main, dtype: int64
```

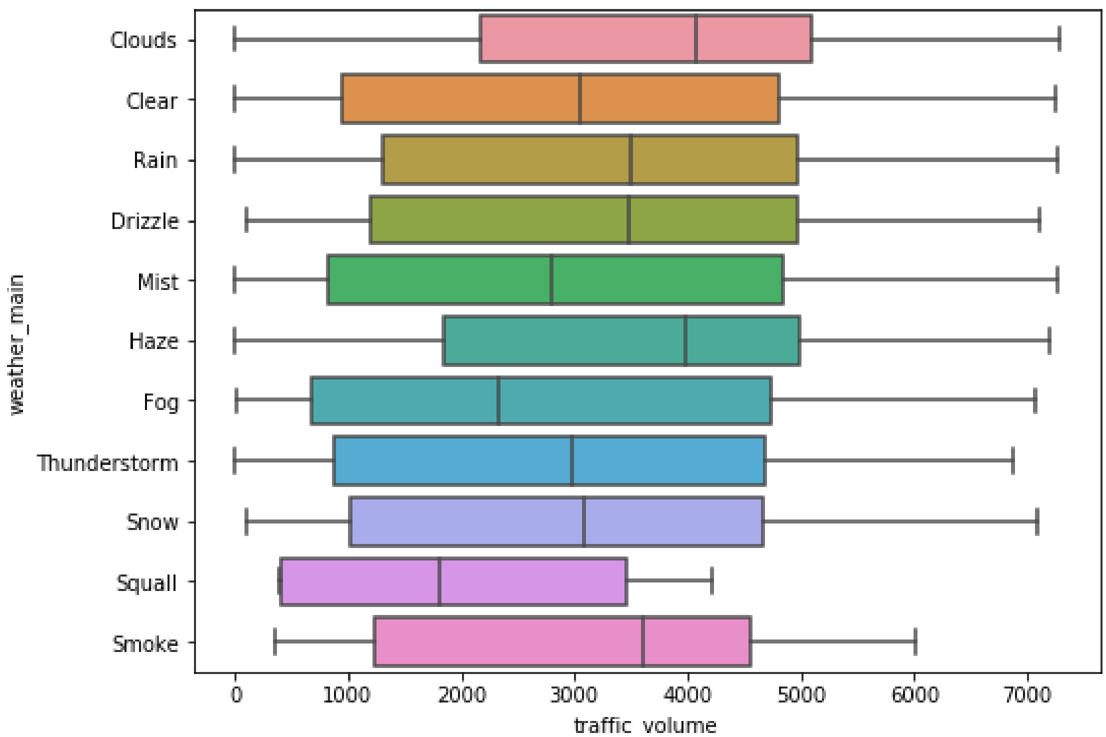
[23]:
```
metro_data['weather_main'].nunique()
```

[23]:
```
11
```

[24]:
```
plt.figure(figsize=(8,6))
sns.countplot(y='weather_main', data= metro_data, order =␣
↪metro_data['weather_main'].value_counts().index)
plt.show()
```



[25]:
```
#Exploring traffic volume on weather
plt.figure(figsize=(8,6))
sns.boxplot(y='weather_main',x='traffic_volume', data = metro_data)
plt.show()
```



Among the various types of weather, the ones that have on average lower traffic volumes are 'Squall' and 'Fog'. The weather for which are registered, on average, major traffic volumes are 'Clouds' and 'Haze'.

2.7 Weather_description

[26]:
```
metro_data['weather_description'].value_counts()
```

[26]:
```
sky is clear                        11665
mist                                 5950
overcast clouds                      5081
broken clouds                        4666
scattered clouds                     3461
light rain                           3372
few clouds                           1956
light snow                           1946
Sky is Clear                         1726
moderate rain                        1664
haze                                 1360
light intensity drizzle              1100
fog                                   912
proximity thunderstorm                673
drizzle                               651
```

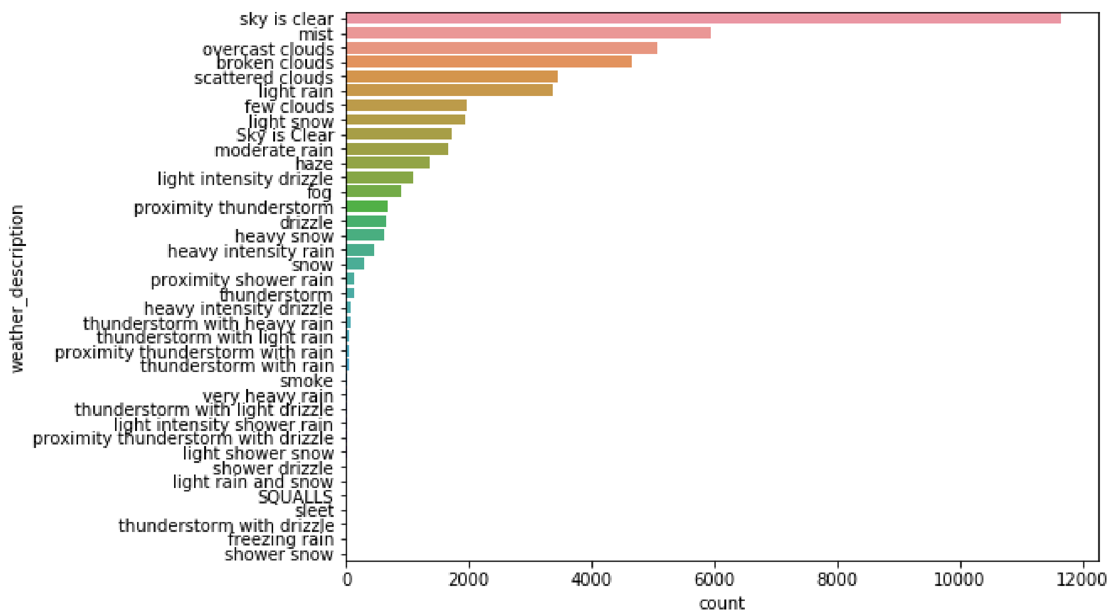
```
heavy snow                              616
heavy intensity rain                    467
snow                                    293
proximity shower rain                   136
thunderstorm                            125
heavy intensity drizzle                  64
thunderstorm with heavy rain             63
thunderstorm with light rain             54
proximity thunderstorm with rain         52
thunderstorm with rain                   37
smoke                                    20
very heavy rain                          18
thunderstorm with light drizzle          15
light intensity shower rain              13
proximity thunderstorm with drizzle      13
light shower snow                        11
shower drizzle                            6
light rain and snow                       6
SQUALLS                                   4
sleet                                     3
thunderstorm with drizzle                 2
freezing rain                             2
shower snow                               1
Name: weather_description, dtype: int64
```

The categories 'sky is clear' and 'sky is Clear' mean the same. This needs to be recoded into same category.

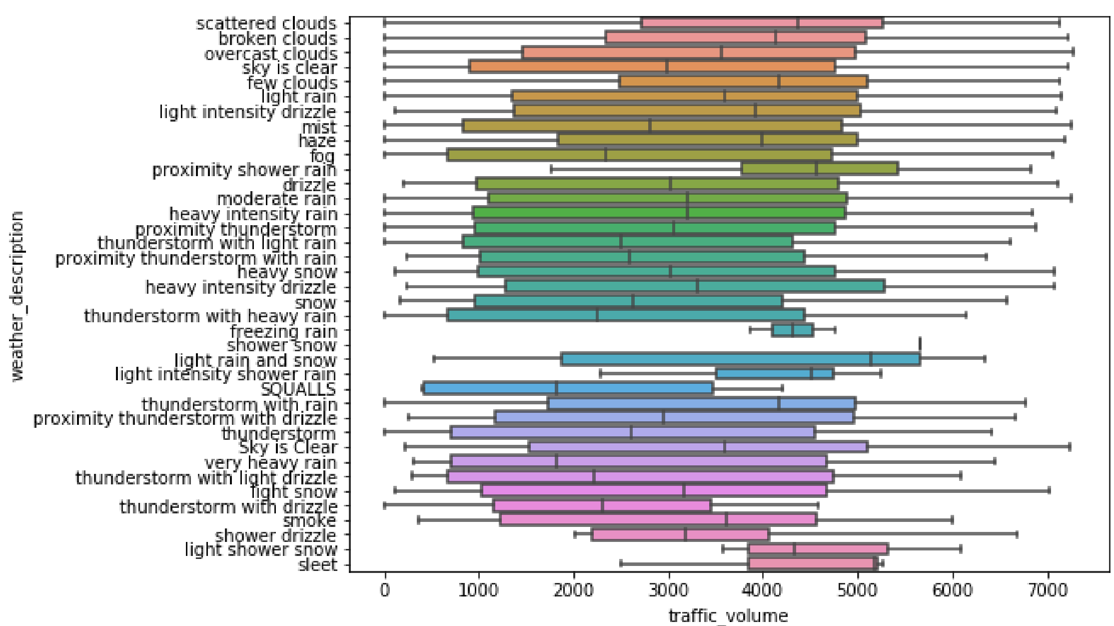
[27]:
```
metro_data['weather_description'].nunique()
```

[27]: 38

[28]:
```
plt.figure(figsize=(8,6))
sns.countplot(y='weather_description', data= metro_data, order =␣
 ↪metro_data['weather_description'].value_counts().index)
plt.show()
```



[29]:
```
#Exploring traffic volume on holidays
plt.figure(figsize=(8,6))
sns.boxplot(y='weather_description',x='traffic_volume', data = metro_data)
plt.show()
```



I decided to remove this variable because the information I could get from it was not strictly necessary, given the variable 'Weather_main' which effectively summarizes the temporal states of the data.

2.8 Date_time

[30]:
```
metro_data['date_time'].min(),metro_data['date_time'].max()
```

[30]:
```
('2012-10-02 09:00:00', '2018-09-30 23:00:00')
```

[31]:
```
metro_data['date_time'].dtypes
```

[31]:
```
dtype('O')
```

The collected data cover a period of 7 years. Thanks to pandas' dtypes function, we know that the date_time variable is still in the object type. So we need to convert it to datetime type. From this feature we can get a lot of information.

2.9 Traffic_volume

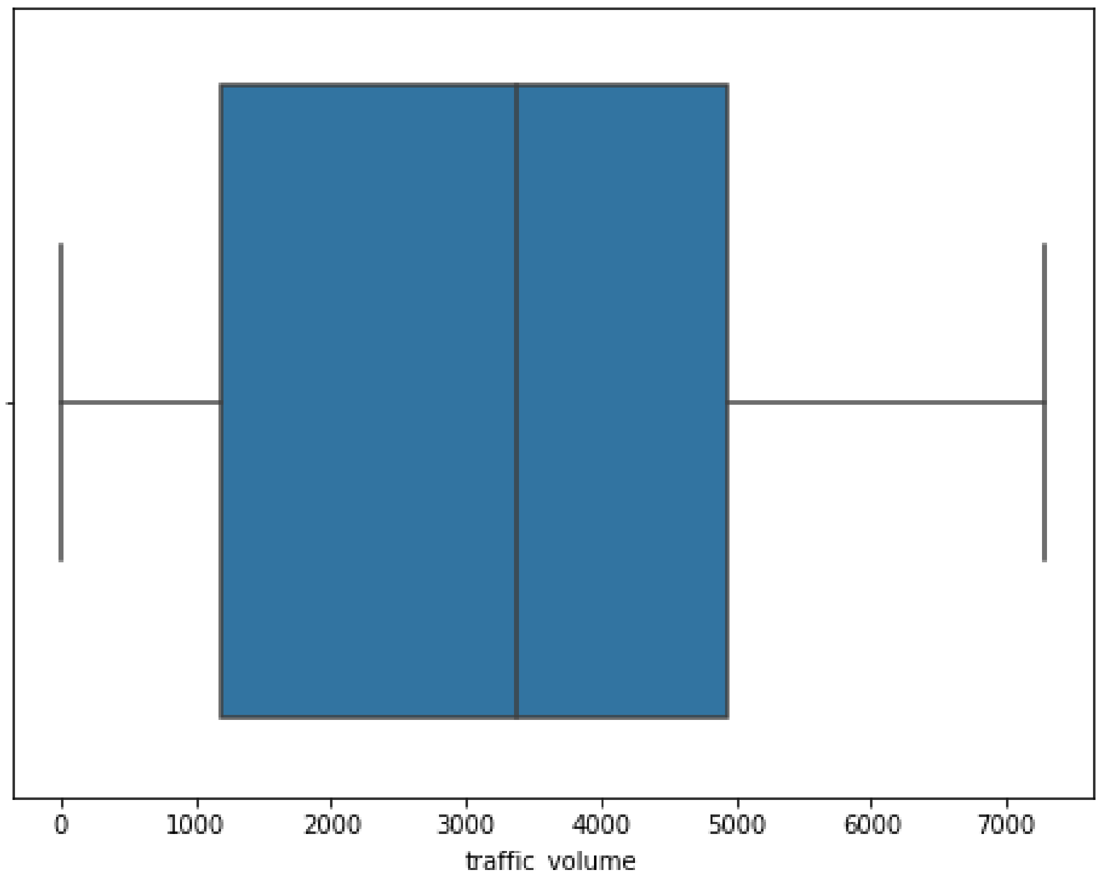
[32]:
```
metro_data['traffic_volume'].describe()
```

[32]:
```
count    48204.000000
mean      3259.818355
std       1986.860670
min          0.000000
25%       1193.000000
50%       3380.000000
75%       4933.000000
max       7280.000000
Name: traffic_volume, dtype: float64
```

[33]:
```
metro_data['traffic_volume'].nunique()
```

[33]:
```
6704
```

[34]:
```
plt.figure(figsize=(8,6))
sns.boxplot('traffic_volume', data = metro_data)
plt.show()
```



3 Data Pre-Processing

During this step I transformed categorical variables into numerical variables resorting to transformations into Boolean variables or with the one-hot-encoding function and I eliminated outliers that were to be attributed to evaluation errors.

3.1 Holiday

[35]:
```
# Change holiday column to be a boolean: 1 if holiday else 0
metro_data["holiday_bool"] = np.where(metro_data.holiday=="None", 0, 1)
```

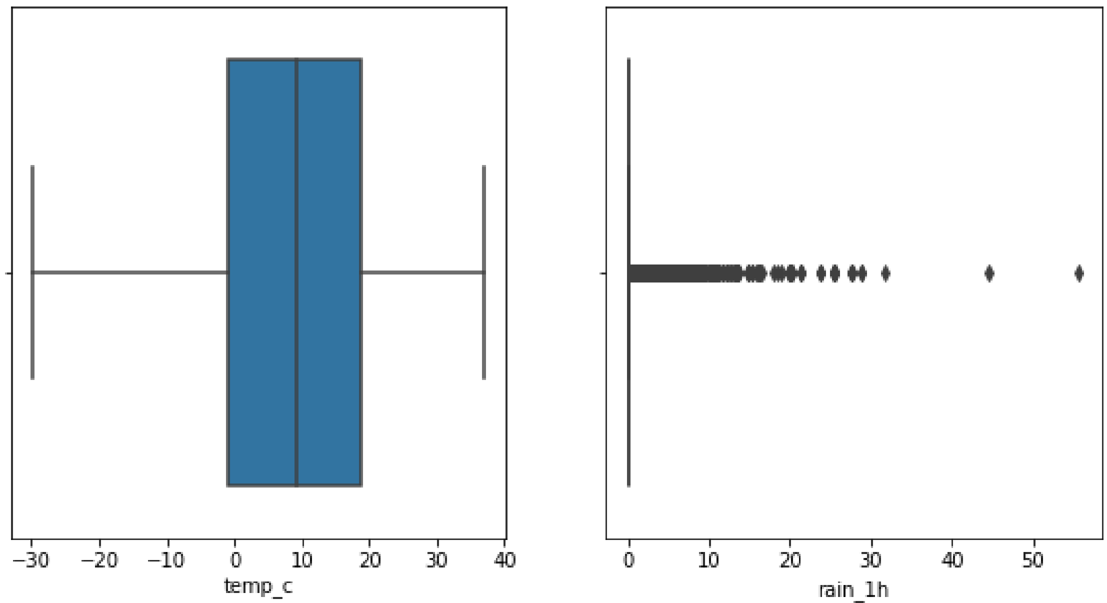
3.2 Temperature and Rain

[36]:
```
# change temp from kelvin to celsius (for better readability)
metro_data['temp_c'] = metro_data['temp'] - 273.15
```

[37]:
```
# drop outliers from temp and rain
metro_data.drop(metro_data[metro_data.temp_c < -50].index, inplace=True)
```

```
metro_data.drop(metro_data[metro_data.rain_1h > 9000].index, inplace=True)
```

[38]:
```
plt.figure(figsize= (10,5))
plt.subplot(1,2,1)
sns.boxplot('temp_c', data = metro_data)
plt.subplot(1,2,2)
sns.boxplot('rain_1h', data = metro_data)
plt.show()
```



3.3 Date_time

[39]:
```
# convert date_time column to datetime type
metro_data.date_time = pd.to_datetime(metro_data.date_time)
```

After transforming the variable into date_time format, we can obtain the years, months, days and hours from it.

3.3.1 Year

[40]:
```
# extract year feature
years = metro_data.date_time.dt.year
years.value_counts()
```

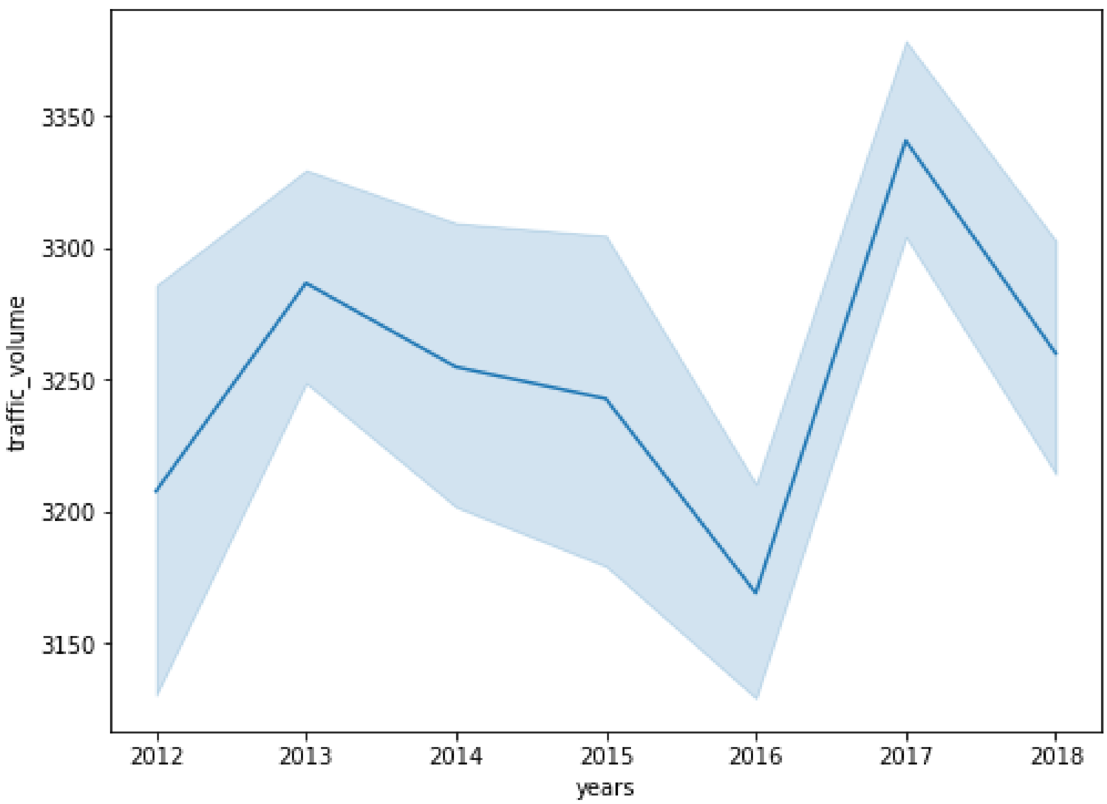
[40]:
```
2017    10605
2016     9305
2013     8573
2018     7949
```

```
2014     4829
2015     4373
2012     2559
Name: date_time, dtype: int64
```

[41]:
```python
time = pd.DataFrame({
    'years' : years,
    'traffic_volume' : metro_data.traffic_volume

    })
```

[42]:
```python
plt.figure(figsize=(8,6))
sns.lineplot('years', 'traffic_volume', data= time)
plt.show()
```



After examining the traffic volume for each year, we can see a decrease in traffic_volume which occurs around the end of 2015 - beginning of 2016. This could also be due to a lack of data collection in that period since during the other years the traffic volume remains rather stable.

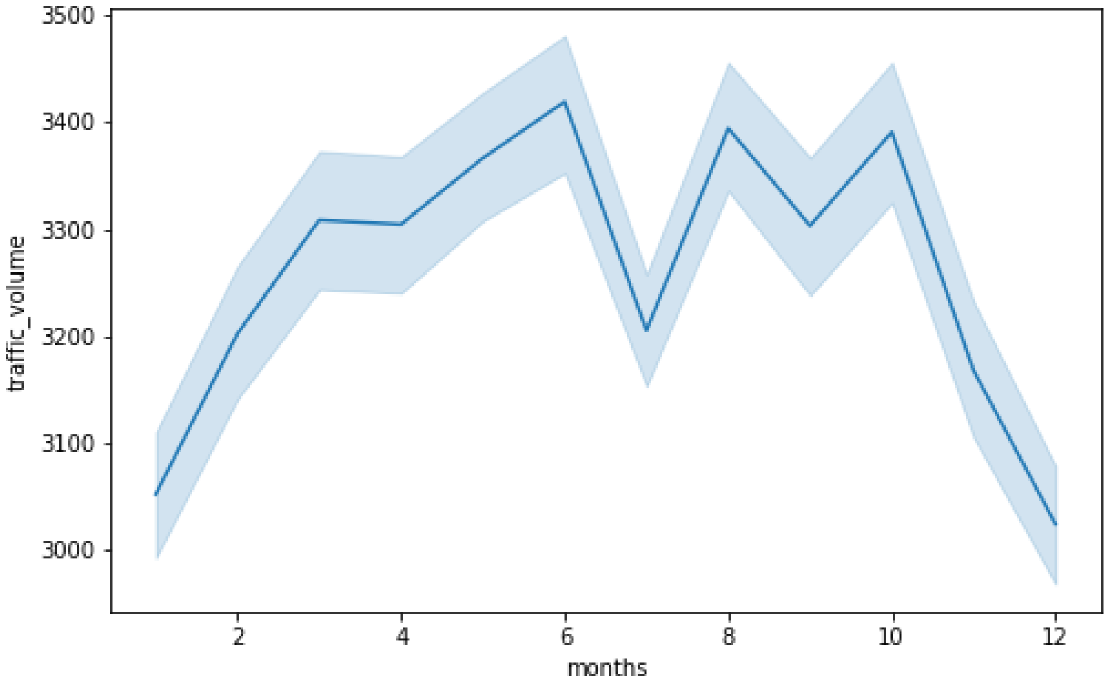
3.3.2 Month

[43]:
```python
# extract month feature
months = metro_data.date_time.dt.month
months.value_counts()
```

[43]:
```
7     4794
5     4436
8     4378
4     4259
12    4249
1     4002
9     3831
3     3793
6     3772
11    3686
2     3520
10    3473
Name: date_time, dtype: int64
```

[44]:
```python
time = pd.DataFrame({
    'years' : years,
    'months': months,
    'traffic_volume' : metro_data.traffic_volume
    })
```

[45]:
```python
plt.figure(figsize=(8,5))
sns.lineplot('months', 'traffic_volume', data= time)
plt.show()
```



The traffic volume begins to grow from January until a positive peak in June. A sharp decrease follows in July. Then there is a new increase during the end of the summer period and it starts to decrease again during the beginning of the winter months.

3.3.3 Day of Month

[46]:
```
# extract day of month feature
day_of_months = metro_data.date_time.dt.day
day_of_months.value_counts()
```

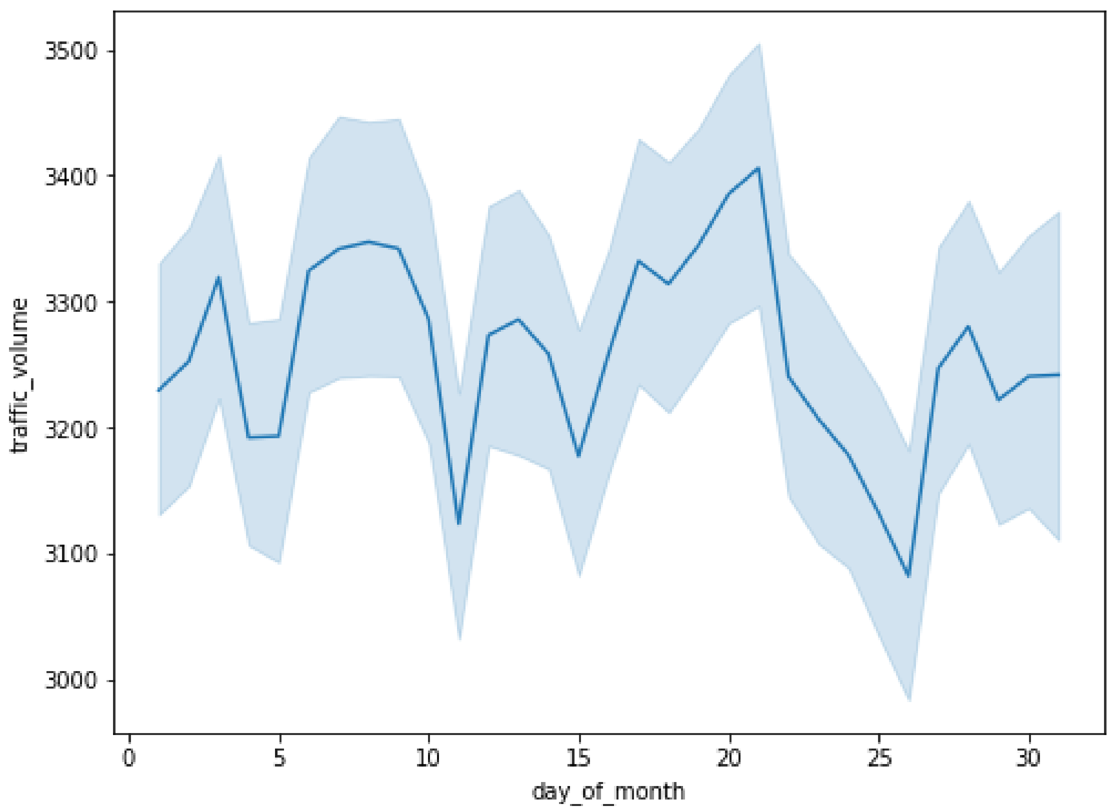
[46]:
```
19      1715
6       1700
11      1660
20      1655
26      1643
25      1640
14      1632
18      1630
16      1627
24      1616
4       1613
15      1610
17      1603
28      1596
23      1595
```

```
10     1594
12     1580
3      1576
9      1573
7      1569
8      1560
5      1541
27     1521
2      1518
22     1516
21     1511
13     1496
1      1443
30     1395
29     1359
31      906
Name: date_time, dtype: int64
```

[47]:
```
time = pd.DataFrame({
    'years' : years,
    'months': months,
    'day_of_month':day_of_months,
    'traffic_volume' : metro_data.traffic_volume

    })
```

[48]:
```
plt.figure(figsize=(8,6))
sns.lineplot('day_of_month', 'traffic_volume', data= time)
plt.show()
```



In this case we notice a rather stable trend in traffic, which remains between the values of 3100 and 3300, with a brief peak towards the end of the month.

3.3.4 Day of Week

This time I process the data differently because the goal is to extract the day name.
The process consists of two steps:
- First is to extract the day name literal using pd.Series.dt.day_name() method.
- Afterwards, we need to one-hot encode the results from the first step using pd.get_dummies() method.

[49]:
```
# first: extract the day name literal
days_name = metro_data.date_time.dt.day_name()
# second: one hot encode to 7 columns
days = pd.get_dummies(days_name)
days = days[['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday',␣
↪'Sunday']]
days
```

[49]:

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0

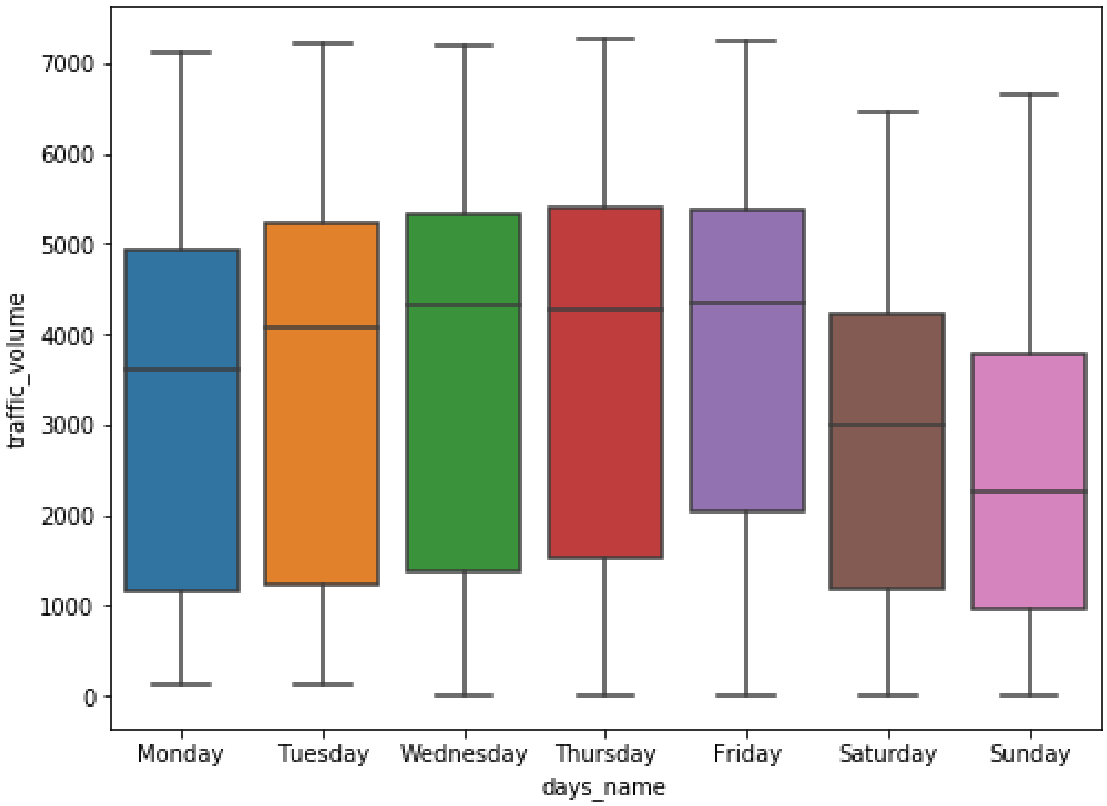
```
2            0        1        0        0        0        0        0
3            0        1        0        0        0        0        0
4            0        1        0        0        0        0        0
…        …        …        …        …        …        …        …
48199        0        0        0        0        0        0        1
48200        0        0        0        0        0        0        1
48201        0        0        0        0        0        0        1
48202        0        0        0        0        0        0        1
48203        0        0        0        0        0        0        1

[48193 rows x 7 columns]
```

[50]:
```python
time = pd.DataFrame({
    'years' : years,
    'months': months,
    'day_of_month':day_of_months,
    'days_name' : days_name,
    'traffic_volume' : metro_data.traffic_volume

    })
```

[51]:
```python
plt.figure(figsize=(8,6))
sns.boxplot(x='days_name',y='traffic_volume', data = time,␣
 ↪order=['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday',␣
 ↪'Sunday'])
plt.show()
```



The traffic volume begins to grow from the first day of the week, Monday, until Friday. During the weekend the volume lowers a lot, especially on Sundays.

3.3.5 Hour

[52]:
```
# extract hour feature
hours = metro_data.date_time.dt.hour
hours.value_counts()
```

[52]:
```
4     2089
6     2085
8     2079
10    2078
7     2078
5     2061
1     2049
23    2040
0     2037
3     2023
2     2019
9     2018
22    1994
```

```
16    1988
18    1986
21    1982
20    1979
14    1969
19    1961
12    1955
11    1952
15    1934
17    1932
13    1905
Name: date_time, dtype: int64
```

This time I will create a grouping based on the hour digits. Six groups representing each daypart: Dawn (02.00 — 05.59), Morning (06.00 —09.59), Noon (10.00–13.59), Afternoon (14.00–17.59), Evening (18.00–21.59), and Midnight (22.00–01.59 on Day+1). To this end, we create an identifying function that we later use to feed an apply method of a Series. Afterwards, we perform one-hot encoding on the resulted dayparts.

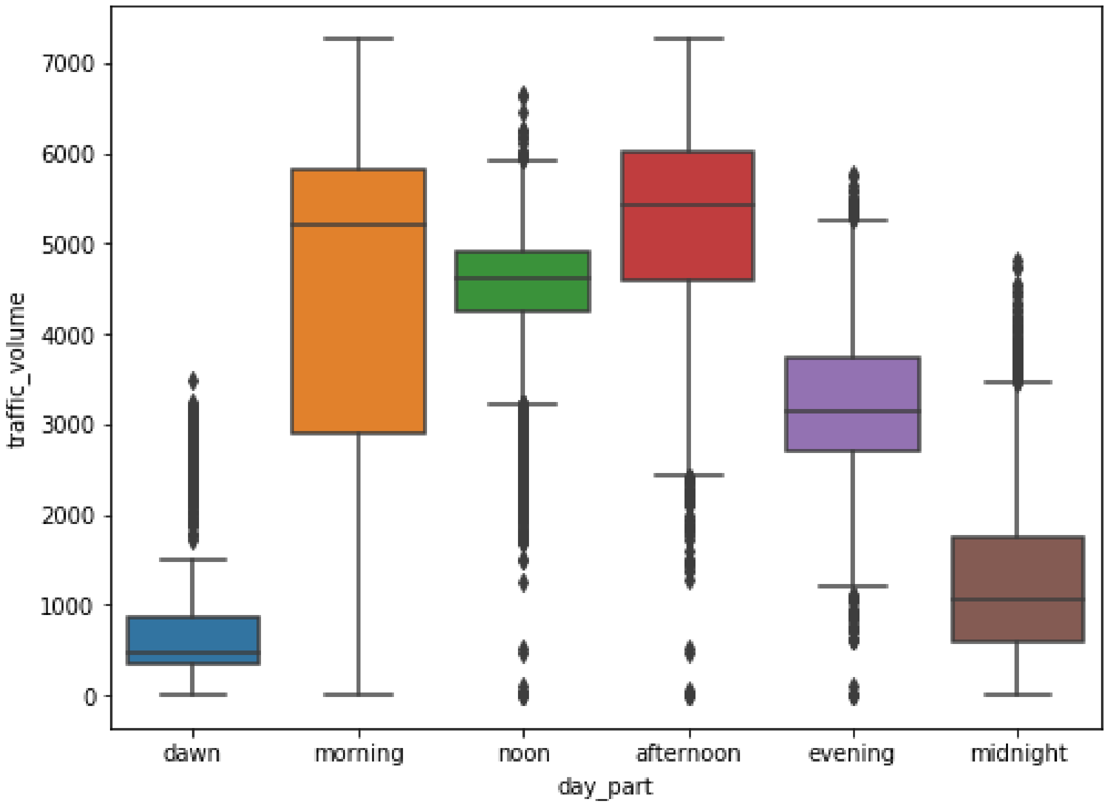
[53]:
```python
# daypart function
def day_part(hours):
    if hours in [2,3,4,5]:
        return "dawn"
    elif hours in [6,7,8,9]:
        return "morning"
    elif hours in [10,11,12,13]:
        return "noon"
    elif hours in [14,15,16,17]:
        return "afternoon"
    elif hours in [18,19,20,21]:
        return "evening"
    else: return "midnight"
```

[54]:
```python
# utilize it along with apply method
day_part = hours.apply(day_part)
```

[55]:
```python
time = pd.DataFrame({
    'years' : years,
    'months': months,
    'day_of_month':day_of_months,
    'days_name' : days_name,
    'day_part' :day_part,
    'traffic_volume' : metro_data.traffic_volume

    })
```

[56]:
```python
plt.figure(figsize=(8,6))
sns.boxplot('day_part', 'traffic_volume', data= time, order=␣
↪['dawn','morning','noon','afternoon','evening','midnight'])
plt.show()
```



The major traffic volume are registerd during morning and afternoon, while very small values are registered during midnight and dawn.

[57]:
```python
# one hot encoding
day_part = pd.get_dummies(day_part)
# re-arrange columns for convenience
day_part = day_part[['dawn','morning','noon','afternoon','evening','midnight']]
#display data
day_part
```

[57]:

	dawn	morning	noon	afternoon	evening	midnight
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	1	0	0	0
3	0	0	1	0	0	0
4	0	0	1	0	0	0

```
…      …         …    …        …        …          …
48199      0         0      0           0          1           0
48200      0         0      0           0          1           0
48201      0         0      0           0          1           0
48202      0         0      0           0          0           1
48203      0         0      0           0          0           1

[48193 rows x 6 columns]
```

3.4 Weather

[58]:
```python
# one-hot encode weather
weathers = pd.get_dummies(metro_data.weather_main)
#display data
weathers
```

[58]:
```
       Clear  Clouds  Drizzle  Fog  Haze  Mist  Rain  Smoke  Snow  Squall  \
0          0       1        0    0     0     0     0      0     0       0
1          0       1        0    0     0     0     0      0     0       0
2          0       1        0    0     0     0     0      0     0       0
3          0       1        0    0     0     0     0      0     0       0
4          0       1        0    0     0     0     0      0     0       0
…      …     …      …   …    …   …   …    …   …     …
48199      0       1        0    0     0     0     0      0     0       0
48200      0       1        0    0     0     0     0      0     0       0
48201      0       0        0    0     0     0     0      0     0       0
48202      0       1        0    0     0     0     0      0     0       0
48203      0       1        0    0     0     0     0      0     0       0

       Thunderstorm
0                 0
1                 0
2                 0
3                 0
4                 0
…              …
48199             0
48200             0
48201             1
48202             0
48203             0

[48193 rows x 11 columns]
```

4 Final Dataset

Finally, I created a new dataset which include all the transformed variables. It will be composed by 48193 rows and 33 columns.

[59]:
```
#features to keep with just one column of values
features = pd.DataFrame({
    'holiday' :metro_data.holiday_bool,
    'temp_c' : metro_data.temp_c,
    'rain_1h' : metro_data.rain_1h,
    'snow_1h' :metro_data.snow_1h,
    'clouds_all' : metro_data.clouds_all,
    'years' : years,
    'months':months,
    'day_of_month' : day_of_months

    })
```

[60]:
```
#concat with one-hot encode typed features
features = pd.concat([features, days,day_part, weathers, metro_data.
 ↪traffic_volume], axis = 1)
```

[61]:
```
features.info()
```

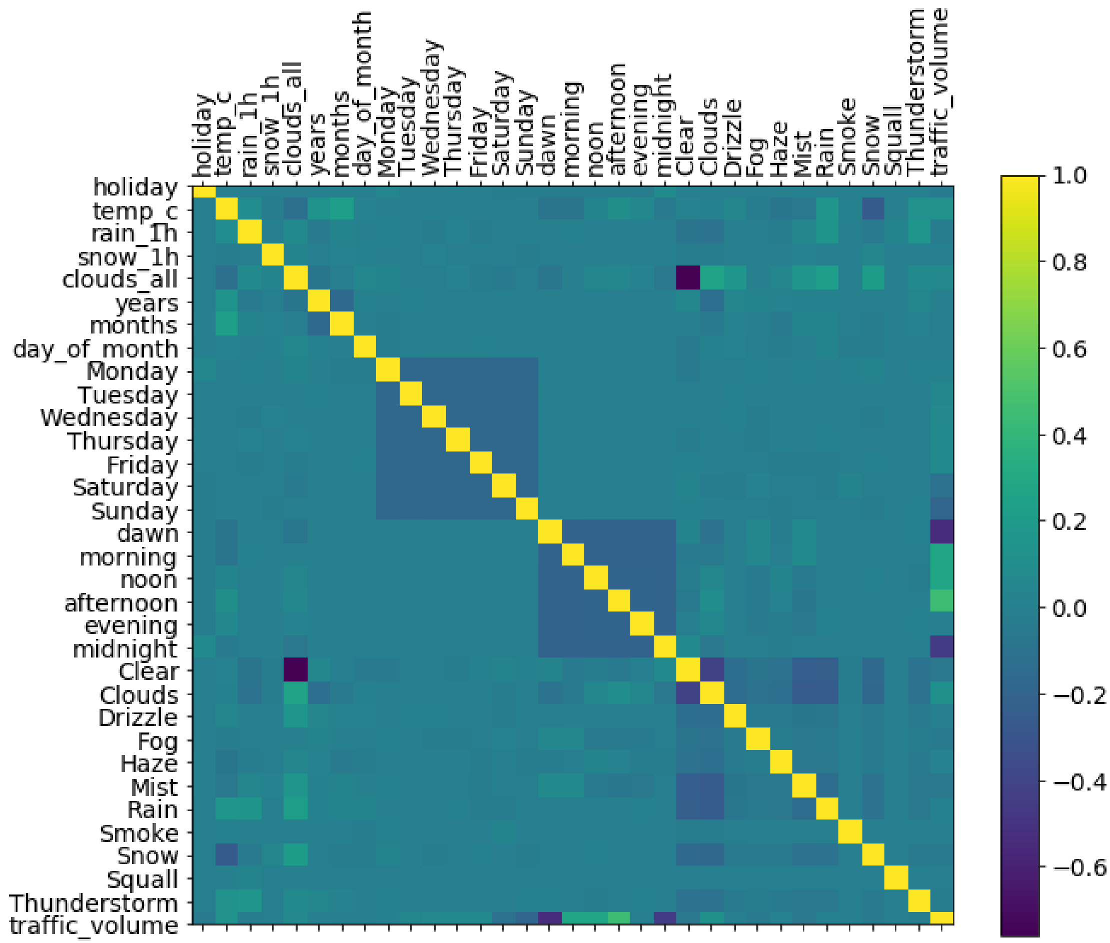
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 48193 entries, 0 to 48203
Data columns (total 33 columns):
holiday          48193 non-null int32
temp_c           48193 non-null float64
rain_1h          48193 non-null float64
snow_1h          48193 non-null float64
clouds_all       48193 non-null int64
years            48193 non-null int64
months           48193 non-null int64
day_of_month     48193 non-null int64
Monday           48193 non-null uint8
Tuesday          48193 non-null uint8
Wednesday        48193 non-null uint8
Thursday         48193 non-null uint8
Friday           48193 non-null uint8
Saturday         48193 non-null uint8
Sunday           48193 non-null uint8
dawn             48193 non-null uint8
morning          48193 non-null uint8
noon             48193 non-null uint8
afternoon        48193 non-null uint8
evening          48193 non-null uint8
midnight         48193 non-null uint8
```

```
Clear              48193 non-null uint8
Clouds             48193 non-null uint8
Drizzle            48193 non-null uint8
Fog                48193 non-null uint8
Haze               48193 non-null uint8
Mist               48193 non-null uint8
Rain               48193 non-null uint8
Smoke              48193 non-null uint8
Snow               48193 non-null uint8
Squall             48193 non-null uint8
Thunderstorm       48193 non-null uint8
traffic_volume     48193 non-null int64
dtypes: float64(3), int32(1), int64(5), uint8(24)
memory usage: 5.8 MB
```

4.1 Correlation Matrix

With all the variables in a numerical type, I can perform the Correlation Matrix.

[62]:
```python
#Correlation Matrix, standard method 'Pearson'
f = plt.figure(figsize=(10, 8))
plt.matshow(features.corr(), fignum=f.number)
plt.xticks(range(features.shape[1]), features.columns, fontsize=14, rotation=90)
plt.yticks(range(features.shape[1]), features.columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
```



There are no strong correlation among all the variables, except the one Clear-Clouds_all and those of the days of the week.
Even more we note the lack of strong correlations between most of the variables and our target variable. The exceptions in this case are to be found on the days of the week (especially those of the weekend) and the part of the day.

4.2 Splitting the dataset

[63]:
```
from sklearn.model_selection import train_test_split # split the dataset into
↪training and test set
from sklearn.metrics import mean_squared_error,r2_score,mean_absolute_error #to
↪evaluate r2 score, mse, mae
```

[64]:
```
X = features.iloc[:,:32]
y = features.iloc[:,32]
```

I decided to split the dataset in training set for the 80% and test set for the last 20%.
Notice below I do not shuffle our data, this is due to the time-series nature of the data.

[65]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪shuffle =False, random_state = 1231)
```

4.3 Normalization

Normalization should be done after splitting the data between training and test set, using only the data from the training set. This is because the testing data points represent real-world data, so it's not supposed to be accessible at the training stage. Using any information coming from the test set before or during training is a potential bias in the evaluation of the performance.

Therefore, we should perform feature scaling over the training data and then perform normalization on testing instances but this time using the mean and standard deviation of training explanatory variables.

[66]:
```
from sklearn.preprocessing import MinMaxScaler

# create a scaler object
scaler = MinMaxScaler()
# fit and transform the data
X_train_norm= pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.
↪columns )
```

[67]:
```
X_test_norm = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns )
```

5 Modelling

5.1 Multiple Linear Regression

Multiple Linear Regression fits a linear model with coefficients to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

[68]:
```
Model= ['Linear Regression', 'Linear SVR', 'SVR','Decision Tree','Random Forest␣
↪Regression', 'Gradient Boosting Regression', 'K-Nearest Neighbors']
R_squared =list()
RMSE = list()
MAE = list()
```

[69]:
```
from sklearn.linear_model import LinearRegression
import sklearn.metrics as metrics
```

[70]:
```
LR = LinearRegression()
LR.fit(X_train_norm,y_train)
```

[70]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[71]:

```
print('R square score on train set and test set are :',LR.
 ↪score(X_train_norm,y_train),LR.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,LR.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,LR.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.7724877134913599
0.7606378100987384
Root mean squared error : 962.7673609135386
Mean absolute error : 742.219246031746
```

[72]:
```
R_squared.append(LR.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,LR.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,LR.predict(X_test_norm)))
```

5.2 Support Vector Machines

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.
The method to solve regression problems is called *Support Vector Regression*, it depends only on a subset of the training data, because the cost function ignores samples whose prediction is close to their target.

[73]:
```
from sklearn import svm
from sklearn.svm import SVR
from sklearn.svm import LinearSVR
```

5.2.1 Linear Support Vector Regression

LinearSVR provides a faster implementation than SVR but only considers the linear kernel, but it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

[74]:
```
LinearSVR = LinearSVR(random_state= 1231)
LinearSVR.fit(X_train_norm,y_train)
```

[74]:
```
LinearSVR(C=1.0, dual=True, epsilon=0.0, fit_intercept=True,
          intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
          random_state=1231, tol=0.0001, verbose=0)
```

[75]:
```
print('R square score on train set and test set are :',LinearSVR.
 ↪score(X_train_norm,y_train),LinearSVR.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,LinearSVR.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,LinearSVR.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.7529013715668929
0.7471637382314086
Root mean squared error : 989.4942258063295
Mean absolute error : 744.0261273800437
```

Tuning the Hyper-parameters During the building of our models, it is possible and recommended to search the hyper-parameter space for the best cross validation score. Any parameter provided when constructing an estimator may be optimized in this manner.
The approach I used to parameter search is provided by **GridSearchCV**, exhaustively generates candidates from a grid of parameter values specified with the parameter_grid parameter.

In this case, I will evaluate models using the negative mean absolute error (neg_mean_absolute_error). It is negative because the GridsearchCV requires the score to be maximized, so the MAE is made negative, meaning scores scale from -infinity to 0 (best).

[76]:
```
from sklearn.model_selection import GridSearchCV
```

[77]:
```
parameter_grid = {'C': range(1,100)}
GS=GridSearchCV(LinearSVR,parameter_grid,cv=3, scoring='neg_mean_squared_error')
GS.fit(X_train_norm,y_train)
```

[77]:
```
GridSearchCV(cv=3, error_score='raise-deprecating',
             estimator=LinearSVR(C=1.0, dual=True, epsilon=0.0,
                                 fit_intercept=True, intercept_scaling=1.0,
                                 loss='epsilon_insensitive', max_iter=1000,
                                 random_state=1231, tol=0.0001, verbose=0),
             iid='warn', n_jobs=None, param_grid={'C': range(1, 100)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_squared_error', verbose=0)
```

[78]:
```
GS.best_params_
```

[78]:
```
{'C': 3}
```

[79]:
```
from sklearn.svm import LinearSVR
```

[80]:
```
HLinearSVR = LinearSVR(C=3, random_state=1213)
HLinearSVR.fit(X_train_norm,y_train)
```

[80]:
```
LinearSVR(C=3, dual=True, epsilon=0.0, fit_intercept=True,
          intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
          random_state=1213, tol=0.0001, verbose=0)
```

[81]:
```
print('R square score on train set and test set are :',HLinearSVR.
 ↪score(X_train_norm,y_train),HLinearSVR.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HLinearSVR.
 ↪predict(X_test_norm))))
```

```
print('Mean absolute error :',mean_absolute_error(y_test,HLinearSVR.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.7521587781157375
0.7414609680490978
Root mean squared error : 1000.5911172338606
Mean absolute error : 714.1834585381343
```

[82]:
```
R_squared.append(HLinearSVR.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HLinearSVR.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HLinearSVR.predict(X_test_norm)))
```

5.2.2 Support Vector Regressor

[83]:
```
SVR = SVR()
SVR.fit(X_train_norm,y_train)
```

[83]:
```
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1,
    gamma='auto_deprecated', kernel='rbf', max_iter=-1, shrinking=True,
    tol=0.001, verbose=False)
```

[84]:
```
print('R square score on train set and test set are :',SVR.
 ↪score(X_train_norm,y_train),SVR.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,SVR.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,SVR.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.2157466197648119
0.2179059989644907
Root mean squared error : 1740.2957567720518
Mean absolute error : 1498.1628361410808
```

Tuning the Hyper-parameters

[85]:
```
parameter_grid = {'C': [1, 10, 100,1000], 'kernel': ['rbf', 'poly']}
GS=GridSearchCV(SVR,parameter_grid,cv=3, scoring='neg_mean_squared_error')
GS.fit(X_train_norm,y_train)
```

[85]:
```
GridSearchCV(cv=3, error_score='raise-deprecating',
             estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
                           epsilon=0.1, gamma='auto_deprecated', kernel='rbf',
                           max_iter=-1, shrinking=True, tol=0.001,
                           verbose=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [1, 10, 100, 1000], 'kernel': ['rbf', 'poly']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_squared_error', verbose=0)
```

[86]:
```
GS.best_params_
```

[86]: {'C': 1000, 'kernel': 'rbf'}

[87]:
```
from sklearn.svm import SVR
```

[88]:
```
HSVR = SVR(C=1000, kernel='rbf')
HSVR.fit(X_train_norm,y_train)
```

[88]:
```
SVR(C=1000, cache_size=200, coef0=0.0, degree=3, epsilon=0.1,
    gamma='auto_deprecated', kernel='rbf', max_iter=-1, shrinking=True,
    tol=0.001, verbose=False)
```

[89]:
```
print('R square score on train set and test set are :',HSVR.
 ↪score(X_train_norm,y_train),HSVR.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HSVR.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,HSVR.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.8477124791915841
0.8313852212688579
Root mean squared error : 808.0558940641218
Mean absolute error : 572.2545017897323
```

[90]:
```
R_squared.append(HSVR.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HSVR.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HSVR.predict(X_test_norm)))
```

5.3 Decision Tree

Decision Trees 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.

[91]:
```
from sklearn.tree import DecisionTreeRegressor
```

[92]:
```
DT = DecisionTreeRegressor(random_state= 1231)
DT.fit(X_train_norm,y_train)
```

[92]:
```
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=1231, splitter='best')
```

[93]:

```
print('R square score on train set and test set are :',DT.
↪score(X_train_norm,y_train),DT.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,DT.
↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,DT.
↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.997213841639757
0.7190681482304146
Root mean squared error : 1043.0234553098385
Mean absolute error : 752.1692602967113
```

5.3.1 Tuning the Hyper-parameters

[94]:
```
parameter_grid = {'max_depth': range(1,25)}
```

[95]:
```
grid_search = GridSearchCV(DT, parameter_grid, cv=3,␣
↪scoring='neg_mean_squared_error')
grid_search.fit(X_train_norm, y_train)
grid_search.best_params_
```

[95]:
```
{'max_depth': 8}
```

[96]:
```
HDT = DecisionTreeRegressor(max_depth=8, random_state=1231)
HDT.fit(X_train_norm,y_train)
```

[96]:
```
DecisionTreeRegressor(criterion='mse', max_depth=8, 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=1231, splitter='best')
```

[97]:
```
print('R square score on train set and test set are :',HDT.
↪score(X_train_norm,y_train),HDT.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HDT.
↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,HDT.
↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.8602568057343928
0.8309474987622335
Root mean squared error : 809.1040672974674
Mean absolute error : 596.9427626404746
```

[98]:
```
R_squared.append(HDT.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HDT.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HDT.predict(X_test_norm)))
```

5.4 Random Forest Regression

Random Forest Regression is a supervised learning algorithm that uses ensemble learning method for regression.
Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

[99]:
```
from sklearn.ensemble import RandomForestRegressor
```

[100]:
```
RF = RandomForestRegressor(random_state= 1231)
RF.fit(X_train_norm,y_train)
```

[100]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features='auto', 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=10,
                      n_jobs=None, oob_score=False, random_state=1231,
                      verbose=0, warm_start=False)
```

[101]:
```
print('R square score on train set and test set are :',RF.
 ↪score(X_train_norm,y_train),RF.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,RF.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,RF.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.979282470553996
0.8369884515463356
Root mean squared error : 794.5162247872779
Mean absolute error : 599.5446410169006
```

5.4.1 Tuning the Hyper-parameters

[102]:
```
parameter_grid ={'max_depth':np.arange(1,25),'n_estimators':np.arange(1,25)}
grid_search = GridSearchCV(RF, parameter_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(X_train_norm, y_train)
grid_search.best_params_
```

[102]:
```
{'max_depth': 11, 'n_estimators': 24}
```

[103]:
```
HRF = RandomForestRegressor(max_depth= 11, n_estimators= 24, random_state= 1231)
HRF.fit(X_train_norm,y_train)
```

[103]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=11,
                      max_features='auto', 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=24,
                             n_jobs=None, oob_score=False, random_state=1231,
                             verbose=0, warm_start=False)
```

[104]:
```python
print('R square score on train set and test set are :',HRF.
 ↪score(X_train_norm,y_train),HRF.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HRF.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,HRF.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.8921345064145638
0.8481992768129891
Root mean squared error : 766.7088993267513
Mean absolute error : 571.1701443787183
```

[105]:
```python
R_squared.append(HRF.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HRF.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HRF.predict(X_test_norm)))
```

5.5 Gradient Boosting Regressor

The Gradient Boosting Machine is a powerful ensemble machine learning algorithm that uses decision trees.
Boosting is a general ensemble technique that involves sequentially adding models to the ensemble where subsequent models correct the performance of prior models.
"Gradient" because it uses a gradient descent algorithm to minimize the loss when adding new models.

[106]:
```python
from sklearn.ensemble import GradientBoostingRegressor
```

[107]:
```python
GB = GradientBoostingRegressor(random_state=1231)
GB.fit(X_train_norm,y_train)
```

[107]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=3,
                          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, n_estimators=100,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

[108]:
```python
print('R square score on train set and test set are :',GB.
 ↪score(X_train_norm,y_train),GB.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,GB.
 ↪predict(X_test_norm))))
```

```
print('Mean absolute error :',mean_absolute_error(y_test,GB.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.8622315201551755
0.8423088468336225
Root mean squared error : 781.442897931951
Mean absolute error : 585.0570613119559
```

5.5.1 Tuning the Hyper-parameters

[109]:
```
parameter_grid ={'max_depth' : [1,5,10,15,20] ,'n_estimators': [10,20,30]}
grid_search = GridSearchCV(GB, parameter_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(X_train_norm, y_train)
grid_search.best_params_
```

[109]:
```
{'max_depth': 10, 'n_estimators': 30}
```

[110]:
```
HGB = GradientBoostingRegressor(max_depth= 10, n_estimators= 30,␣
 ↪random_state=1231)
HGB.fit(X_train_norm,y_train)
```

[110]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=10,
                          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, n_estimators=30,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

[111]:
```
print('R square score on train set and test set are :',HGB.
 ↪score(X_train_norm,y_train),HGB.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HGB.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,HGB.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.9013101417007605
0.8489734455433307
Root mean squared error : 764.7513303653811
Mean absolute error : 581.2252549663989
```

[112]:
```
R_squared.append(HGB.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HGB.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HGB.predict(X_test_norm)))
```

5.6 K- Nearest Neighbors

K-Nearest Neighbors is a simple algorithm that stores all available cases and predict the numerical target based on a similarity measure (e.g., distance functions). For regression the algorithm just takes the mean of the nearest k neighbors.

[113]:
```
from sklearn.neighbors import KNeighborsRegressor
```

[114]:
```
KN = KNeighborsRegressor()
KN.fit(X_train_norm,y_train)
```

[114]:
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                    weights='uniform')
```

[115]:
```
print('R square score on train set and test set are :',KN.
 ↪score(X_train_norm,y_train),KN.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,KN.
 ↪predict(X_test_norm))))
print('Mean absolute error :',mean_absolute_error(y_test,KN.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.8982259603519164
0.8175531994341717
Root mean squared error : 840.5464337273637
Mean absolute error : 619.9348895113601
```

5.6.1 Tuning the Hyper-parameters

[116]:
```
parameter_grid ={'n_neighbors' : np.arange(1,20) ,'weights': ['uniform',␣
 ↪'distance']}
grid_search = GridSearchCV(KN, parameter_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(X_train_norm, y_train)
grid_search.best_params_
```

[116]:
```
{'n_neighbors': 19, 'weights': 'distance'}
```

[117]:
```
HKN = KNeighborsRegressor(n_neighbors= 19, weights= 'distance')
HKN.fit(X_train_norm,y_train)
```

[117]:
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=19, p=2,
                    weights='distance')
```

[118]:
```
print('R square score on train set and test set are :',HKN.
 ↪score(X_train_norm,y_train),HKN.score(X_test_norm,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HKN.
 ↪predict(X_test_norm))))
```

```
print('Mean absolute error :',mean_absolute_error(y_test,HKN.
 ↪predict(X_test_norm)))
```

```
R square score on train set and test set are : 0.997213841639757
0.834422970845402
Root mean squared error : 800.743877933003
Mean absolute error : 590.9703858374536
```

[119]:
```
R_squared.append(HKN.score(X_test_norm, y_test))
RMSE.append(np.sqrt(mean_squared_error(y_test,HKN.predict(X_test_norm))))
MAE.append(mean_absolute_error(y_test,HKN.predict(X_test_norm)))
```

5.7 Comparing the results

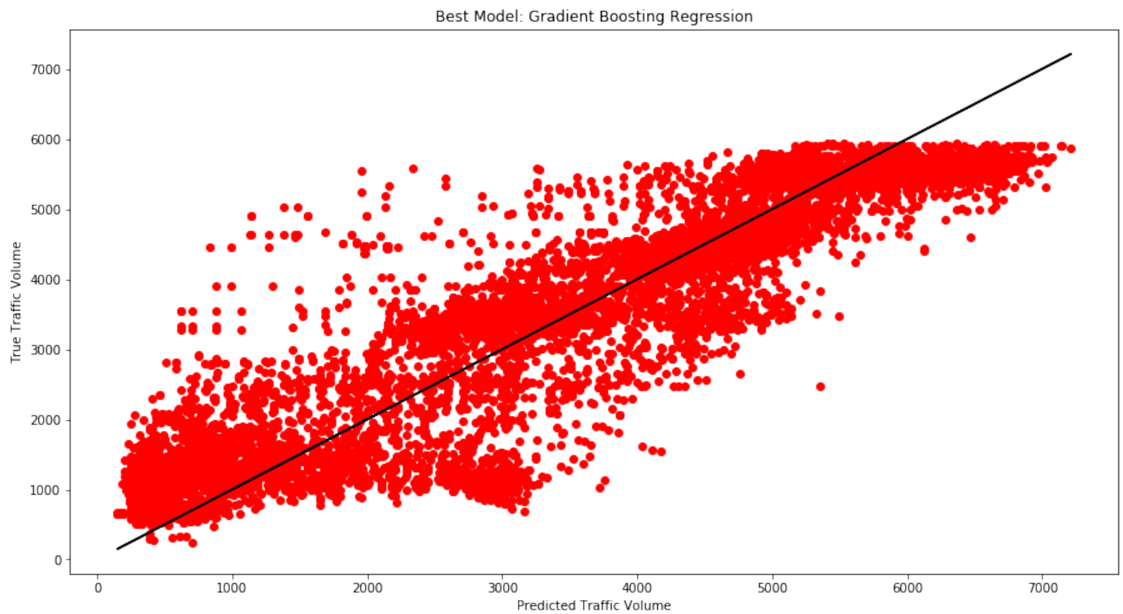
The metrics on which I have chosen the models are:
• **R Squared**: measures how much of variability in dependent variable can be explained by the model. Value are between 0 to 1 and bigger value indicates a better fit between prediction and actual value.
• **Mean Absolute Error (MAE)** : It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight. It is negatively-oriented scores, which means lower values are better.
• **Root Mean Squared Error (RMSE)**: is the square root of the variance of the residuals. It indicates the absolute fit of the model to the data, how close the observed data points are to the model's predicted values. Lower values of RMSE indicate better fit. RMSE is a good measure of how accurately the model predicts the response, and it is the most important criterion for fit if the main purpose of the model is prediction.

[120]:
```
score = {"Model": Model, "R_squared": R_squared,"MAE": MAE,"RMSE": RMSE}
df_score = pd.DataFrame(score)
df_score
```

[120]:

	Model	R_squared	MAE	RMSE
0	Linear Regression	0.760638	742.219246	962.767361
1	Linear SVR	0.741461	714.183459	1000.591117
2	SVR	0.831385	572.254502	808.055894
3	Decision Tree	0.830947	596.942763	809.104067
4	Random Forest Regression	0.848199	571.170144	766.708899
5	Gradient Boosting Regression	0.848973	581.225255	764.751330
6	K-Nearest Neighbors	0.834423	590.970386	800.743878

[121]:
```
plt.figure(figsize=(15,8))
plt.scatter(y_test,HGB.predict(X_test_norm),color = "red",Label = "Scatter")
plt.plot(y_test,y_test,color = "black")
plt.xlabel("Predicted Traffic Volume")
plt.ylabel("True Traffic Volume")
plt.title("Best Model: Gradient Boosting Regression")
plt.show()
```



6 Principal Component Analysis

6.1 Dimensionality reduction

Principal Component Analysis (PCA) is an unsupervised linear transformation technique that can be utilized for extracting information from a high-dimensional space by projecting it into a lower-dimensional sub-space. It tries to preserve the essential parts that have more variation of the data and remove the non-essential parts with fewer variation.

If we use PCA for dimensionality reduction, we construct a d x k–dimensional transformation matrix W that allows us to map a sample vector x onto a new k–dimensional feature subspace that has fewer dimensions than the original d–dimensional feature space. As a result, the first principal component will have the largest possible variance, and all consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other principal components.

Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features prior to PCA.

[122]:
```
from sklearn.decomposition import PCA
# Make an instance of the Model
pca = PCA(.90)
```

Let's pass 0.9 as a parameter to the PCA model, which means that PCA will hold 90% of the variance and the number of components required to capture 90% variance will be used.

[123]:
```
#Fit PCA on training set
```

```
pca.fit(X_train_norm)
```

[123]:
```
PCA(copy=True, iterated_power='auto', n_components=0.9, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

[124]:
```
X_train_norm.shape
```

[124]:
```
(38554, 32)
```

[125]:
```
pca.n_components_
```

[125]:
```
17
```

From the above output, you can observe that to achieve 90% variance, the dimension was reduced to 17 principal components from the actual 32 dimensions. Finally, I apply transform on both the training and test set to generate a transformed dataset from the parameters generated from the fit method.

[126]:
```
PCA_X_train = pca.transform(X_train_norm)
PCA_X_test = pca.transform(X_test_norm)
```

6.2 Modeling using PCA

[127]:
```
def train_test_various_models(models, PCA_X_train, PCA_X_test, y_train, y_test):
    """trains and test given models on given data"""
    for model_name, model in models.items():
        model.fit(PCA_X_train, y_train)
        R_squared_PCA.append(model.score(PCA_X_test, y_test))
        RMSE_PCA.append(np.sqrt(mean_squared_error(y_test,model.
 ↪predict(PCA_X_test))))
        MAE_PCA.append(mean_absolute_error(y_test,model.predict(PCA_X_test)))
```

[128]:
```
Model_PCA= ['Linear Regression','Linear SVR','SVR','Decision Tree','Random␣
 ↪Forest Regression', 'Gradient Boosting Regression', 'K-Nearest Neighbors']
R_squared_PCA =list()
RMSE_PCA = list()
MAE_PCA = list()
```

[129]:
```
from sklearn.svm import LinearSVR
from sklearn.svm import SVR
```

[130]:
```
# train and test various models using all features

models = {
    'Linear Regression': LinearRegression(),
    'Linear SVR' : LinearSVR(random_state= 1231),
    'Support Vector Regressor' : SVR(),
    'Decision Tree' : DecisionTreeRegressor(random_state= 1231),
```

```
    'Random Forest Regression': RandomForestRegressor(random_state= 1231),
    'Gradient Boosting Regression':␣
 ↪GradientBoostingRegressor(random_state=1231),
    'Nearest Neighbors': KNeighborsRegressor()

}

train_test_various_models(models, PCA_X_train, PCA_X_test, y_train, y_test)
```

[131]:
```
score_PCA = {"Model": Model_PCA, "R_squared": R_squared_PCA,"MAE":␣
 ↪MAE_PCA,"RMSE": RMSE_PCA}
df_score_PCA = pd.DataFrame(score_PCA)
df_score_PCA
```

[131]:

	Model	R_squared	MAE	RMSE
0	Linear Regression	0.757018	745.386752	970.020710
1	Linear SVR	0.746614	739.077471	990.569230
2	SVR	0.328823	1372.408519	1612.175001
3	Decision Tree	0.636955	861.136166	1185.697466
4	Random Forest Regression	0.788545	683.523279	904.903768
5	Gradient Boosting Regression	0.823532	620.783058	826.660224
6	K-Nearest Neighbors	0.818244	617.143168	838.953788

6.3 Tuning the Hyper-parameters of the best model

[132]:
```
GB_PCA = GradientBoostingRegressor(random_state=1231)
GB_PCA.fit(PCA_X_train,y_train)
```

[132]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=3,
                          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, n_estimators=100,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

[133]:
```
parameter_grid ={'max_depth' : [1,5,10,15,20] ,'n_estimators': [10,20,30]}
grid_search = GridSearchCV(GB_PCA, parameter_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(PCA_X_train, y_train)
grid_search.best_params_
```

[133]:
```
{'max_depth': 5, 'n_estimators': 30}
```

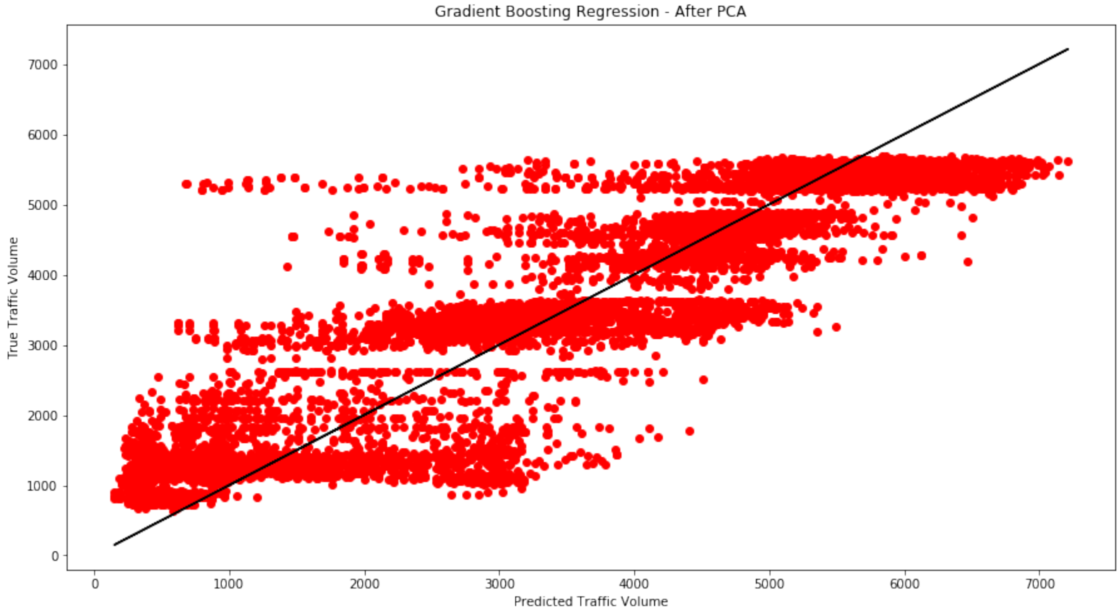
[134]:
```
HGB_PCA = GradientBoostingRegressor(max_depth= 5, n_estimators= 30,␣
    ↪random_state=1231)
HGB_PCA.fit(PCA_X_train,y_train)
```

[134]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=5,
                          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, n_estimators=30,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

[135]:
```
print('R square score on train set and test set are :',HGB_PCA.
    ↪score(PCA_X_train,y_train),HGB_PCA.score(PCA_X_test,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HGB_PCA.
    ↪predict(PCA_X_test))))
print('Mean absolute error :',mean_absolute_error(y_test,HGB_PCA.
    ↪predict(PCA_X_test)))
```

```
R square score on train set and test set are : 0.8490434876663793
0.8219329035768603
Root mean squared error : 830.3963326430467
Mean absolute error : 633.4408514101087
```

[136]:
```
plt.figure(figsize=(15,8))
plt.scatter(y_test,HGB_PCA.predict(PCA_X_test),color = "red",Label = "Scatter")
plt.plot(y_test,y_test,color = "black")
plt.xlabel("Predicted Traffic Volume")
plt.ylabel("True Traffic Volume")
plt.title("Gradient Boosting Regression - After PCA")
plt.show()
```



Although PCA has reduced the computation time of our models, we cannot say that the accuracy results have improved if we compare them with with the results obtained without PCA. On the contrary, they have worsened slightly.

7 Feature Selection

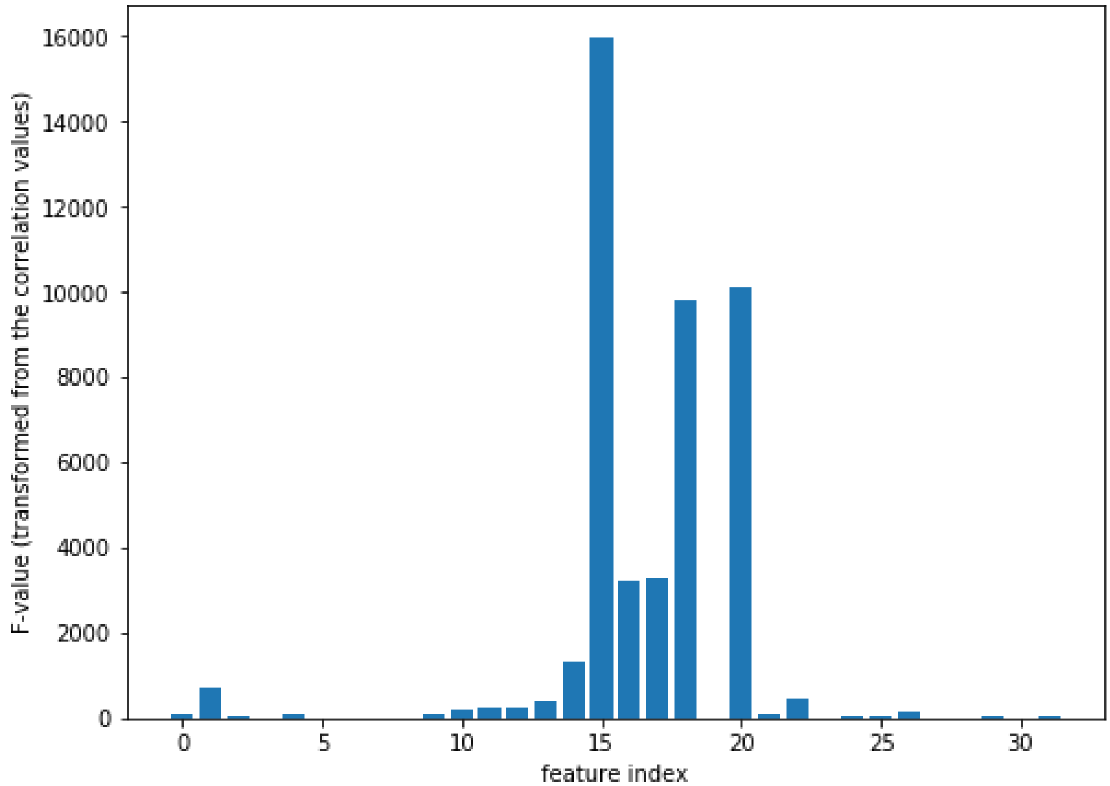
Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable in which you are interested in. The positive aspects of using this technique are:
• Reduces Overfitting: Less redundant data means less opportunity to make decisions based on noise.
• Improves Accuracy: Less misleading data means modeling accuracy improves.
• Reduces Training Time: fewer data points reduce algorithm complexity and algorithms train faster.

The feature selection method I used is 'SelectKBest' using as score function the *f_regression*. The number of top features to select is 15. I'll fit and transform the model on training x and y data.

[137]:
```
from sklearn.feature_selection import SelectKBest, f_regression
from matplotlib import pyplot
```

[138]:
```
fs = SelectKBest(score_func=f_regression, k=15)
fs.fit(X_train_norm, y_train)
X_train_fs = fs.transform(X_train_norm)
X_test_fs = fs.transform(X_test_norm)
```

```
# Plot the scores for the features
plt.figure(figsize=(8,6))
plt.bar([i for i in range(len(fs.scores_))], fs.scores_)
plt.xlabel("feature index")
plt.ylabel("F-value (transformed from the correlation values)")
plt.show()
```



The plot above shows that feature 15 and 20 are more important than the other features. The y-axis represents the F-values that were estimated from the correlation values.

[139]:
```
fs_df = pd.DataFrame({'Feature':list(X_train_norm.columns),
                                        'Scores':fs.scores_})
fs_df.sort_values(by='Scores', ascending=False)
```

[139]:

	Feature	Scores
15	dawn	15943.934472
20	midnight	10093.376477
18	afternoon	9780.316891
17	noon	3266.456411
16	morning	3223.294637
14	Sunday	1324.226311
1	temp_c	708.227254

```
22          Clouds    441.664928
13        Saturday    394.901729
12          Friday    230.664228
11        Thursday    230.619037
10       Wednesday    175.086986
26            Mist    127.189310
21           Clear    110.962201
4       clouds_all    108.090806
9          Tuesday     80.569073
0          holiday     70.305456
24             Fog     32.562107
2          rain_1h     31.367995
29            Snow     27.196711
25            Haze     25.193458
31    Thunderstorm     24.158409
8           Monday      8.865253
7     day_of_month      5.898078
6           months      3.030473
27            Rain      2.234889
5            years      2.021657
30          Squall      1.451608
23         Drizzle      0.298284
28           Smoke      0.112831
19         evening      0.056160
3          snow_1h      0.024352
```

[140]:

```python
filter = fs.get_support()
variable = X_train_norm.columns

print("All features:")
print(variable)

print("Selected best 15:")
print(variable[filter])
```

```
All features:
Index(['holiday', 'temp_c', 'rain_1h', 'snow_1h', 'clouds_all', 'years',
       'months', 'day_of_month', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
       'Friday', 'Saturday', 'Sunday', 'dawn', 'morning', 'noon', 'afternoon',
       'evening', 'midnight', 'Clear', 'Clouds', 'Drizzle', 'Fog', 'Haze',
       'Mist', 'Rain', 'Smoke', 'Snow', 'Squall', 'Thunderstorm'],
      dtype='object')
Selected best 15:
Index(['temp_c', 'clouds_all', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
       'Sunday', 'dawn', 'morning', 'noon', 'afternoon', 'midnight', 'Clear',
       'Clouds', 'Mist'],
      dtype='object')
```

[141]:
```python
print ("Before performing feature selection:" , X_train_norm.shape)
```

```
Before performing feature selection: (38554, 32)
```

[142]:
```python
print ("After selecting best 15 features:",  X_train_fs.shape)
```

```
After selecting best 15 features: (38554, 15)
```

7.1 Modeling after Feature Selection

[143]:
```python
def train_test_various_models(models, X_train_fs, X_test_fs, y_train, y_test):
    """trains and test given models on given data"""
    for model_name, model in models.items():
        model.fit(X_train_fs, y_train)
        R_squared_FS.append(model.score(X_test_fs, y_test))
        RMSE_FS.append(np.sqrt(mean_squared_error(y_test,model.
 ↪predict(X_test_fs))))
        MAE_FS.append(mean_absolute_error(y_test,model.predict(X_test_fs)))
```

[144]:
```python
Model_FS= ['Linear Regression',
            'Linear SVR','SVR',
            'Decision Tree','Random Forest Regression', 'Gradient Boosting␣
 ↪Regression', 'K-Nearest Neighbors']
R_squared_FS =list()
RMSE_FS = list()
MAE_FS = list()
```

[145]:
```python
models = {
    'Linear Regression': LinearRegression(),
    'Linear SVR' : LinearSVR(random_state= 1231),
    'Support Vector Regressor' : SVR(),
    'Decision Tree' : DecisionTreeRegressor(random_state= 1231),
    'Random Forest Regression': RandomForestRegressor(random_state= 1231),
    'Gradient Boosting Regression':␣
 ↪GradientBoostingRegressor(random_state=1231),
    'Nearest Neighbors': KNeighborsRegressor()

}

train_test_various_models(models, X_train_fs, X_test_fs, y_train, y_test)
```

[146]:
```python
score_FS = {"Model": Model_FS, "R_squared": R_squared_FS,"MAE": MAE_FS,"RMSE":␣
 ↪RMSE_FS}
df_score_FS = pd.DataFrame(score_FS)
df_score_FS
```

[146]:

	Model	R_squared	MAE	RMSE
0	Linear Regression	0.760186	742.100859	963.675348

```
1                  Linear SVR   0.741584   750.630699  1000.353484
2                         SVR   0.374213  1328.874650  1556.706891
3               Decision Tree   0.699671   760.368019  1078.430191
4    Random Forest Regression   0.793183   650.258465   894.925981
5  Gradient Boosting Regression 0.836246   590.185080   796.323825
6          K-Nearest Neighbors  0.806093   629.067476   866.542881
```

7.2 Tuning the Hyper-parameters of the best model

[147]:
```
GB_FS= GradientBoostingRegressor(random_state=1231)
GB_FS.fit(X_train_fs,y_train)
```

[147]:
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=3,
                          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, n_estimators=100,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

[148]:
```
print('R square score on train set and test set are :',GB_FS.
 ↪score(X_train_fs,y_train),GB_FS.score(X_test_fs,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,GB_FS.
 ↪predict(X_test_fs))))
print('Mean absolute error :',mean_absolute_error(y_test,GB_FS.
 ↪predict(X_test_fs)))
```

```
R square score on train set and test set are : 0.849941509840402
0.8362458740402997
Root mean squared error : 796.3238252400317
Mean absolute error : 590.1850804651516
```

[149]:
```
parameter_grid ={'max_depth' : [1,5,10,15,20] ,'n_estimators': [10,20,30,50]}
grid_search = GridSearchCV(GB_FS, parameter_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error')
grid_search.fit(X_train_fs, y_train)
grid_search.best_params_
```

[149]:
```
{'max_depth': 5, 'n_estimators': 50}
```

[150]:
```
HGB_FS = GradientBoostingRegressor(max_depth= 5, n_estimators= 50,␣
 ↪random_state=1231)
HGB_FS.fit(X_train_fs,y_train)
```

[150]: GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                          learning_rate=0.1, loss='ls', max_depth=5,
                          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, n_estimators=50,
                          n_iter_no_change=None, presort='auto',
                          random_state=1231, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)

[151]:
```
print('R square score on train set and test set are :',HGB_FS.
 ↪score(X_train_fs,y_train),HGB_FS.score(X_test_fs,y_test))
print('Root mean squared error :',np.sqrt(mean_squared_error(y_test,HGB_FS.
 ↪predict(X_test_fs))))
print('Mean absolute error :',mean_absolute_error(y_test,HGB_FS.
 ↪predict(X_test_fs)))
```

```
R square score on train set and test set are : 0.8593006148511753
0.841423528854804
Root mean squared error : 783.6334366166549
Mean absolute error : 579.4385821059943
```

[152]:
```
plt.figure(figsize=(15,8))
plt.scatter(y_test,GB_FS.predict(X_test_fs),color = "red",Label = "Scatter")
plt.plot(y_test,y_test,color = "black")
plt.xlabel("Predicted Traffic Volume")
plt.ylabel("True Traffic Volume")
plt.title("Gradient Boosting Regression - After Feature Selection")
plt.show()
```

8 Conclusion

Between all the models I performed, the best result is obtained in correspondence of the Gradient Boosting Regression, after performing the Tuning of Hyperparameters. In particular, the selected model is formed by a **max_dept of 5** and a **number of estimators equal to 10**.

Despite this, even the model obtained after performing the **Feature selection** achieves excellent results, slightly lower (RMSE 764.75 vs. 783.63) than one which takes into account all the variables. But we must consider that thanks to the feature selection we use 15 variables instead of 32 and this allowed us to have a faster performance.

In my opinion, this model can be useful to the popolation of Minneapolis and St Paul, in Minnesota, because it's possible to elaborate a strategy according to their Department of Transportation to avoid the creation of massive traffic volume.
Some strategy could be to:

- Improve communication with citizens so that they are aware of the most critical periods;
- Suggest alternative streets during the most trafficate period;
- Increase the control of security and traffic management officers.