Bike rental

By Mohamed Jamyl

http://linkedin.com/in/mohamed-jamyl

https://www.kaggle.com/mohamedjamyl

https://github.com/Mohamed-Jamyl

In [1]:
```
from IPython.display import Image
Image(filename='bi.png')
```

Out[1]:

Project Overview

Bike share systems are a way to rent bikes where the process of obtaining membership, renting and returning bikes is automated through a network of kiosks located throughout the city. Through these systems, people can rent a bike from one location and return it to another location based on their needs. Currently, there are more than 500 bike share programs around the world.

Import Libraries

In [2]:
```
from pandas import read_csv, DataFrame, concat, melt
from matplotlib.pyplot import show, suptitle, subplots_adjust, tight_layout, subplots, scatter
from matplotlib.pyplot import figure, title, xlabel, ylabel, grid, tight_layout
from numpy import log, inf
from seaborn import kdeplot, heatmap, boxplot, regplot, countplot ,barplot, pointplot
from datetime import datetime
import calendar
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from xgboost import XGBRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR

from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from math import sqrt
from pickle import dump


import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Exploratory Data Analysis (EDA)

Initial Data Understanding

- **Data loading and Inspection**
- **Data Types**
- **Missing Values**
- **Duplicates**

In [3]:
```
df = read_csv("vlib.csv")
df.head()
```

Out[3]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

In [4]:
```
df.shape
```

Out[4]:
```
(10886, 12)
```

In [5]:
```
df.columns
```

Out[5]:
```
Index(['datetime', 'season', 'holiday', 'workingday', 'weather', 'temp',
       'atemp', 'humidity', 'windspeed', 'casual', 'registered', 'count'],
      dtype='object')
```

In [6]:
```
df.rename(columns={'casual': 'non-subscribed_users',
                   'registered':'subscribed_users',
                   'count': 'count_of_users',
                   'temp':'Actual_Temperature',
                   'atemp':'Feels_Like_Temperature'} ,inplace=True)
```

In [7]:
```
df
```

Out[7]:

	datetime	season	holiday	workingday	weather	Actual_Temperature	Feels_Like_Temperature	humidity	windspeed	subsc
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0000	
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0000	
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0000	
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0000	
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0000	
...	...	...	...	...	...	...	...	...	...	
10881	2012-12-19 19:00:00	4	0	1	1	15.58	19.695	50	26.0027	
10882	2012-12-19 20:00:00	4	0	1	1	14.76	17.425	57	15.0013	
10883	2012-12-19 21:00:00	4	0	1	1	13.94	15.910	61	15.0013	
10884	2012-12-19 22:00:00	4	0	1	1	13.94	17.425	61	6.0032	
10885	2012-12-19 23:00:00	4	0	1	1	13.12	16.665	66	8.9981	

10886 rows × 12 columns

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   datetime                10886 non-null  object
 1   season                  10886 non-null  int64
 2   holiday                 10886 non-null  int64
 3   workingday              10886 non-null  int64
 4   weather                 10886 non-null  int64
 5   Actual_Temperature      10886 non-null  float64
 6   Feels_Like_Temperature  10886 non-null  float64
 7   humidity                10886 non-null  int64
 8   windspeed               10886 non-null  float64
 9   non-subscribed_users    10886 non-null  int64
 10  subscribed_users        10886 non-null  int64
 11  count_of_users          10886 non-null  int64
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

In [9]:
```
df.isnull().sum()
```

Out[9]:
```
datetime                  0
season                    0
holiday                   0
workingday                0
weather                   0
Actual_Temperature        0
Feels_Like_Temperature    0
humidity                  0
windspeed                 0
non-subscribed_users      0
subscribed_users          0
count_of_users            0
dtype: int64
```

In [10]:
```
df.duplicated().sum()
```

Out[10]: 0

Basic Statistical Overview

- Summary Statistical : **describe()**

In [11]:
```
df.describe().T
```

Out[11]:

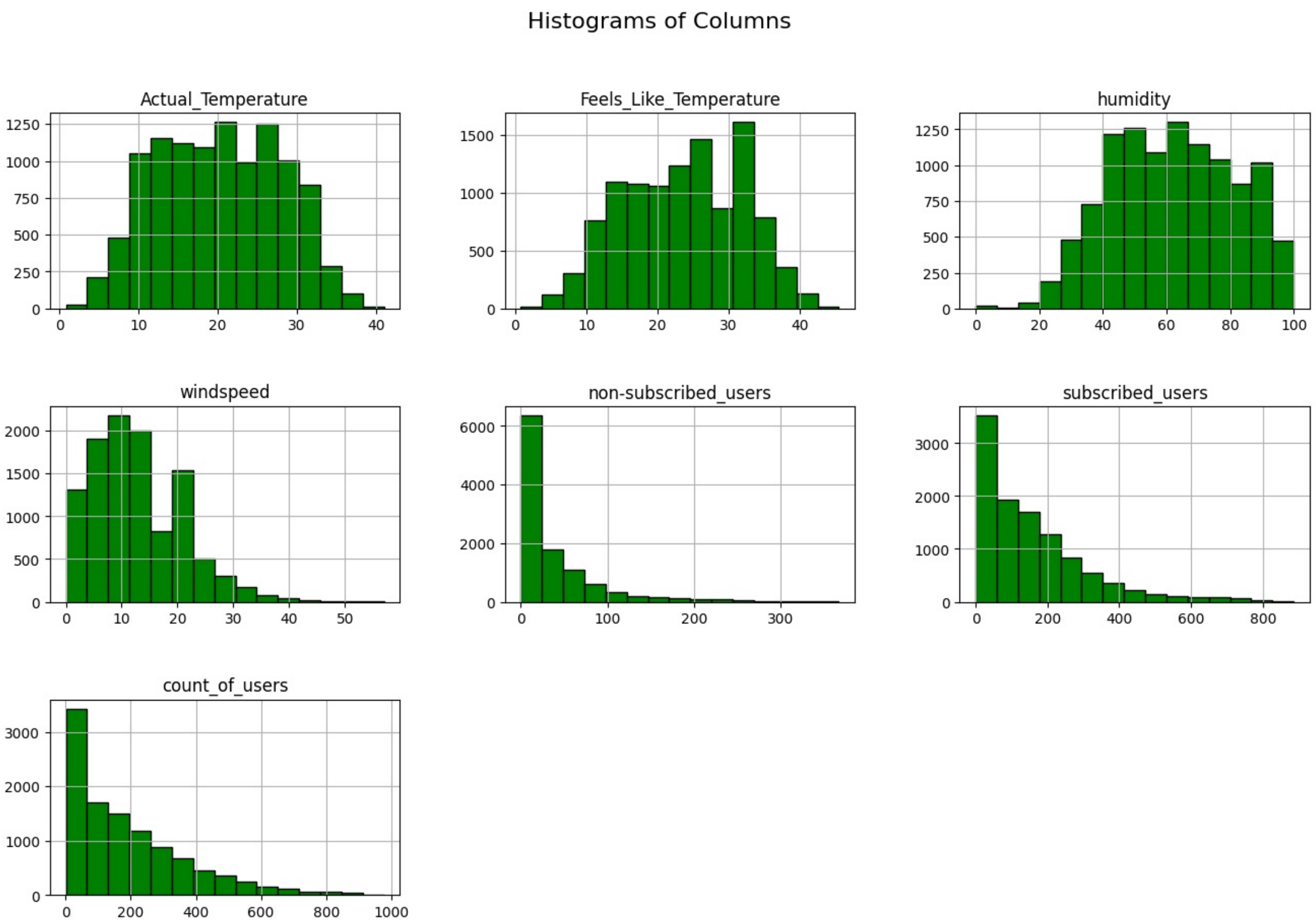
	count	mean	std	min	25%	50%	75%	max
season	10886.0	2.506614	1.116174	1.00	2.0000	3.000	4.0000	4.0000
holiday	10886.0	0.028569	0.166599	0.00	0.0000	0.000	0.0000	1.0000
workingday	10886.0	0.680875	0.466159	0.00	0.0000	1.000	1.0000	1.0000
weather	10886.0	1.418427	0.633839	1.00	1.0000	1.000	2.0000	4.0000
Actual_Temperature	10886.0	20.230860	7.791590	0.82	13.9400	20.500	26.2400	41.0000
Feels_Like_Temperature	10886.0	23.655084	8.474601	0.76	16.6650	24.240	31.0600	45.4550
humidity	10886.0	61.886460	19.245033	0.00	47.0000	62.000	77.0000	100.0000
windspeed	10886.0	12.799395	8.164537	0.00	7.0015	12.998	16.9979	56.9969
non-subscribed_users	10886.0	36.021955	49.960477	0.00	4.0000	17.000	49.0000	367.0000
subscribed_users	10886.0	155.552177	151.039033	0.00	36.0000	118.000	222.0000	886.0000
count_of_users	10886.0	191.574132	181.144454	1.00	42.0000	145.000	284.0000	977.0000

In [12]:
```
df.select_dtypes(include='object').describe()
```

Out[12]:

	datetime
count	10886
unique	10886
top	2011-01-01 00:00:00
freq	1

In [13]:
```
df.drop(['datetime','season','holiday','workingday','weather'],axis=1).hist(bins=15, figsize=(16, 10), color='g

# Set titles and labels for each subplot
suptitle('Histograms of Columns', fontsize=16)
subplots_adjust(hspace=0.5)  # Add space between plots
show()
```

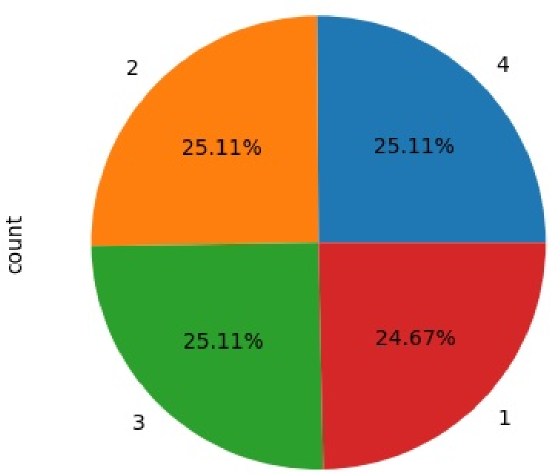


- Summary Statistical : **value_counts()**

In [14]:
```
df['season'].value_counts()
```

Out[14]:
```
season
4    2734
2    2733
3    2733
1    2686
Name: count, dtype: int64
```

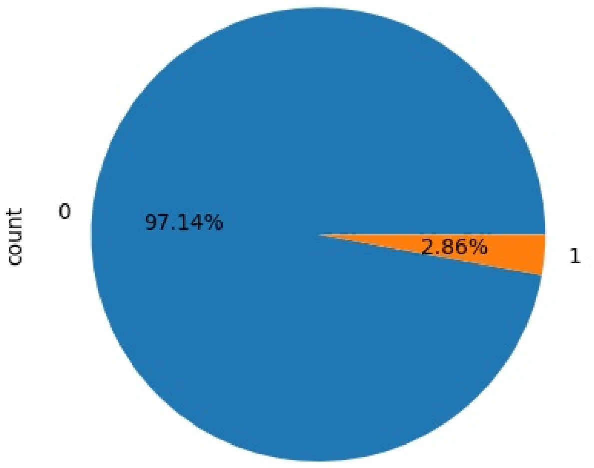
In [15]:
```
df['season'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



In [16]:
```
df['holiday'].value_counts()
```

Out[16]:
```
holiday
0    10575
1      311
Name: count, dtype: int64
```

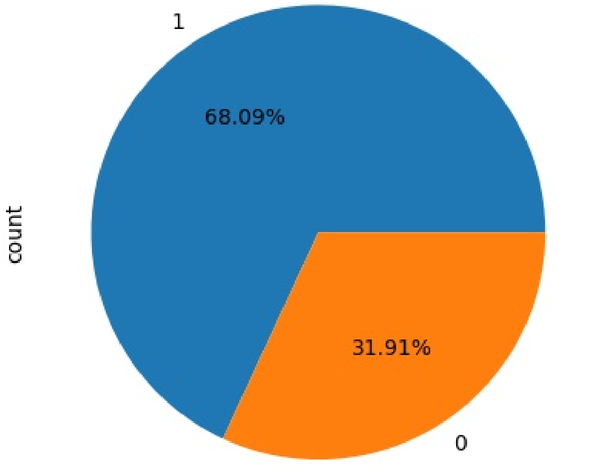
In [17]:
```
df['holiday'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



In [18]:
```
df['workingday'].value_counts()
```

Out[18]:
```
workingday
1    7412
0    3474
Name: count, dtype: int64
```

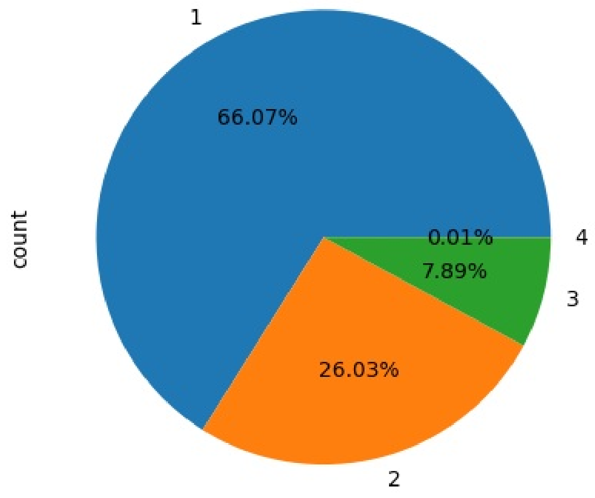
In [19]:
```
df['workingday'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



In [20]:
```
df['weather'].value_counts()
```

Out[20]:
```
weather
1    7192
2    2834
3     859
4       1
Name: count, dtype: int64
```

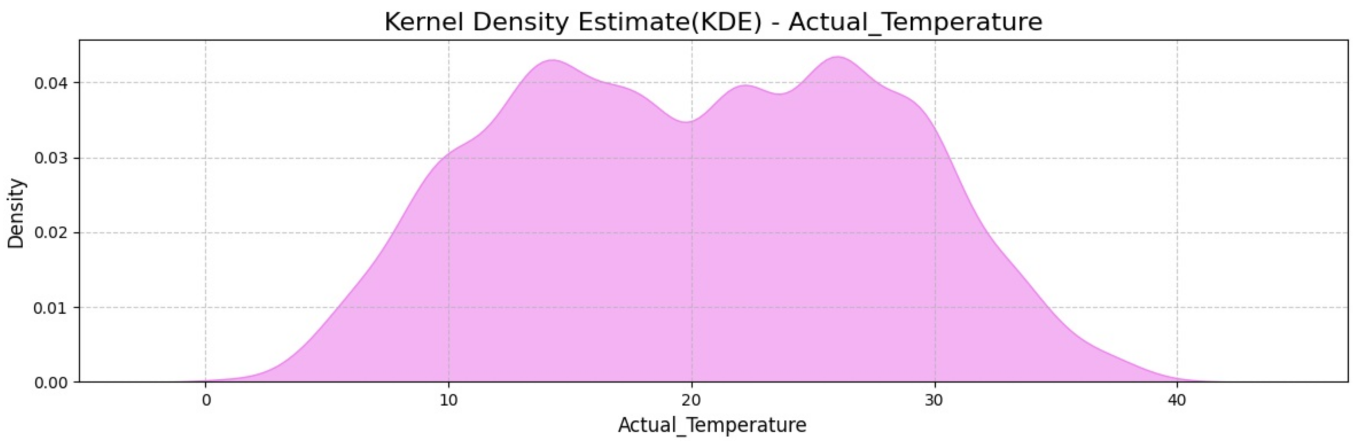
In [21]:
```
df['weather'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



Distribution of Variables

- **Numerical Features (KDE)**

In [22]:
```
figure(figsize=(12,     4))
kdeplot(df['Actual_Temperature'], fill=True,    color='violet', alpha=0.6)
title(f'Kernel Density Estimate(KDE) - Actual_Temperature',     fontsize=16)
xlabel('Actual_Temperature', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



Shape of the Distribution: The distribution of 'Actual Temperature' appears to be bimodal, meaning it has two distinct peaks.

- The first peak is roughly around 14-15 degrees.
- The second, slightly higher peak, is around 25-26 degrees.
- The distribution is somewhat symmetrical around these two peaks, but with a slight tendency to extend further on the right side.

Concentration of Data:

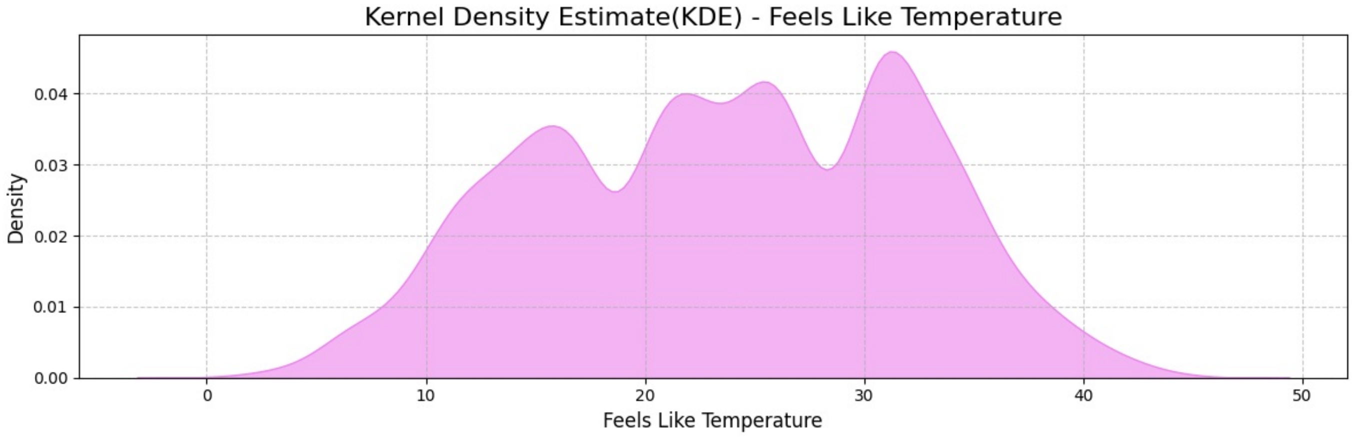
- There are two main clusters of actual temperatures, corresponding to the two peaks. This suggests that the temperatures frequently fall into these two distinct ranges.
- The density is lower at the very low (near 0 degrees) and very high (above 40 degrees) ends of the temperature scale, indicating that these extreme temperatures are less common.

Range of Temperatures: The temperatures range approximately from just below 0 degrees to slightly over 40 degrees.

The KDE plot for 'Actual Temperature' reveals a bimodal distribution, suggesting two common temperature ranges. This could potentially indicate a dataset that combines temperatures from different

seasons (e.g., cooler and warmer periods) or different geographical locations with distinct temperature profiles.

In [23]:
```
figure(figsize=(12,     4))
kdeplot(df['Feels_Like_Temperature'], fill=True,        color='violet', alpha=0.6)
title(f'Kernel Density Estimate(KDE) - Feels Like Temperature', fontsize=16)
xlabel('Feels Like Temperature', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



Shape of the Distribution: The distribution of 'Feels Like Temperature' appears to be multi-modal, specifically with three discernible peaks.

- The first peak is roughly around 14-15 degrees.
- The second peak is around 25 degrees.
- The third, and highest, peak is around 32 degrees.
- The distribution is generally spread across the range, with a slight tendency to extend further on the right side.

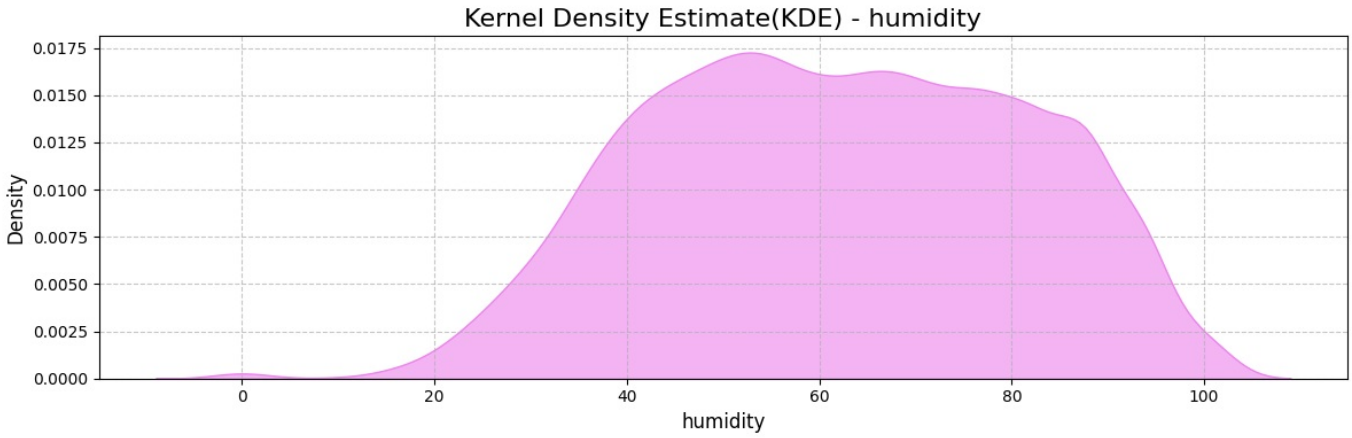
Concentration of Data:

- There are three main clusters of 'Feels Like Temperature' values, corresponding to the three peaks. This suggests that the "feels like" temperatures frequently fall into these three distinct ranges.
- The density is lower at the very low (near 0 degrees) and very high (above 40 degrees) ends of the temperature scale, indicating that these extreme "feels like" temperatures are less common.

Range of Temperatures: The 'Feels Like Temperature' values range approximately from just below 0 degrees to slightly over 40 degrees.

The KDE plot for 'Feels Like Temperature' reveals a multi-modal distribution (three peaks), suggesting multiple common "feels like" temperature ranges. This could potentially indicate a dataset that combines temperatures from different seasons (e.g., cooler, mild, and warmer periods) or different geographical locations with distinct perceived temperature profiles.

In [24]:
```
figure(figsize=(12,     4))
kdeplot(df['humidity'], fill=True,      color='violet', alpha=0.6)
title(f'Kernel Density Estimate(KDE) - humidity',       fontsize=16)
xlabel('humidity', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



Shape of the Distribution: The distribution of 'humidity' appears to be bimodal, meaning it has two distinct peaks.

- The first, smaller peak, is roughly around 55% humidity.
- The second, slightly higher peak, is around 80% humidity.
- The distribution is generally symmetrical around these two peaks, but with a slight tendency to extend further on the right side.

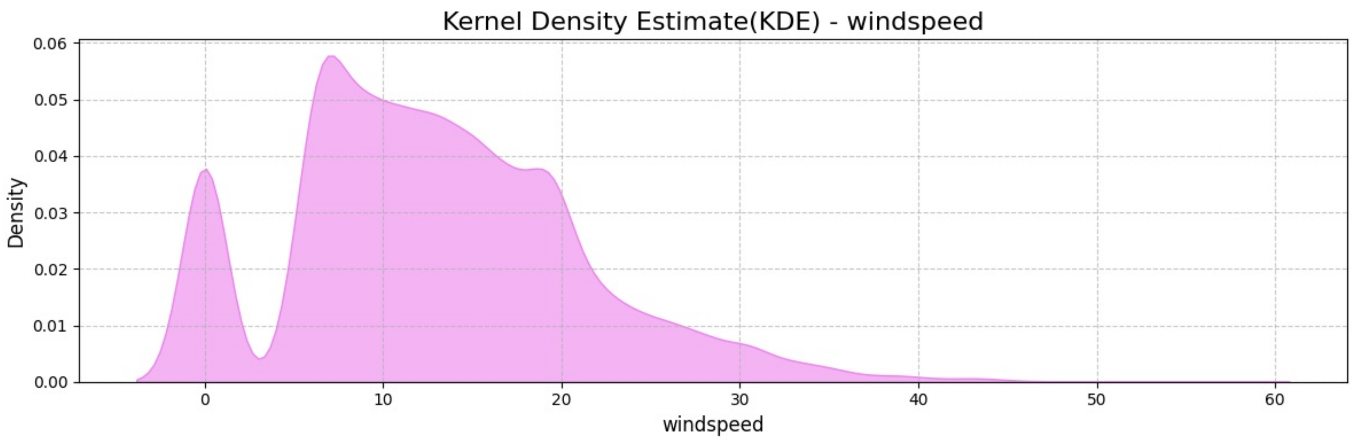
Concentration of Data:

- There are two main clusters of humidity values, corresponding to the two peaks. This suggests that humidity frequently falls into these two distinct ranges.
- The density is very low at extremely low humidity levels (near 0-20%) and gradually tapers off at very high humidity levels (above 90%, reaching close to 100%).

Range of Humidity: The humidity values range approximately from just below 0% to slightly over 100%.

The KDE plot for 'humidity' reveals a bimodal distribution, suggesting two common humidity ranges. This could potentially indicate a dataset that combines data from different environmental conditions (e.g., drier and more humid periods, or different locations with distinct humidity profiles).

In [25]:
```
figure(figsize=(12,     4))
kdeplot(df['windspeed'], fill=True,     color='violet', alpha=0.6)
title(f'Kernel Density Estimate(KDE) - windspeed',      fontsize=16)
xlabel('windspeed', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



Shape of the Distribution: The distribution of 'windspeed' appears to be multi-modal, with at least two distinct peaks.

- The first, smaller peak, is very close to 0 (zero) windspeed.
- The second, and highest, peak is around 7-8 windspeed units.
- There's also a noticeable bump or third peak around 18-20 windspeed units.
- The distribution is right-skewed, with a long tail extending towards higher windspeeds (up to around 60 units).

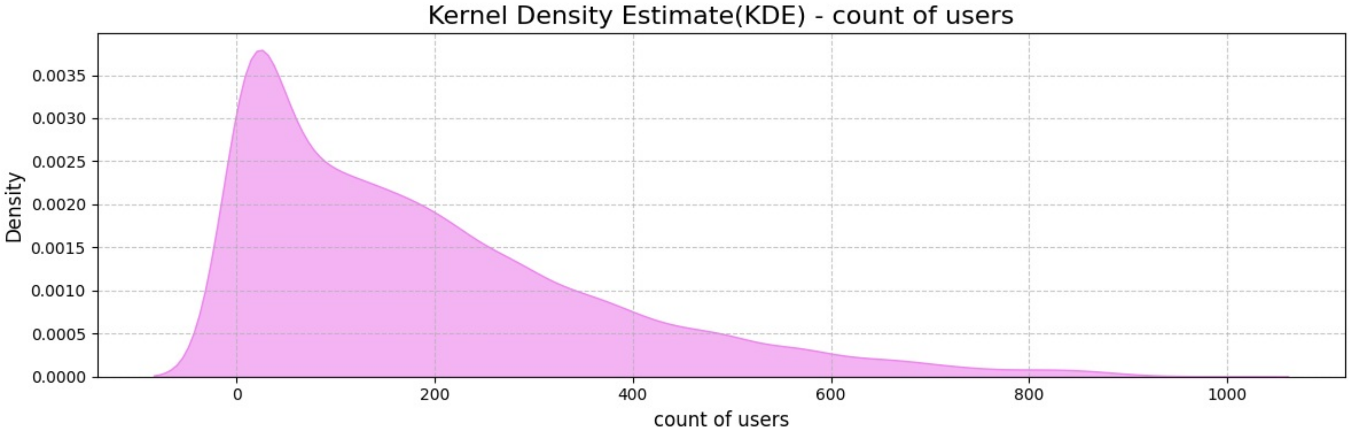
Concentration of Data:

- A significant portion of the data is concentrated at very low windspeeds (near zero), suggesting periods of calm or very light winds.
- The most frequent windspeeds are in the 7-8 unit range.
- Another common range for windspeed is around 18-20 units.
- Higher windspeeds (above 20 units) become progressively less common, as indicated by the tapering tail of the distribution.

Range of Windspeeds: The windspeed values range approximately from 0 to just over 60 units.

The KDE plot for 'windspeed' reveals a complex distribution with multiple common windspeed ranges. The most frequent windspeeds are moderate (around 7-8 units), but there are also many instances of very low windspeeds and a smaller, but still noticeable, cluster of higher windspeeds. The right-skewness indicates that extremely high windspeeds are rare.

In [26]:
```
figure(figsize=(12,     4))
kdeplot(df['count_of_users'], fill=True,        color='violet', alpha=0.6)
title(f'Kernel Density Estimate(KDE) - count of users', fontsize=16)
xlabel('count of users', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



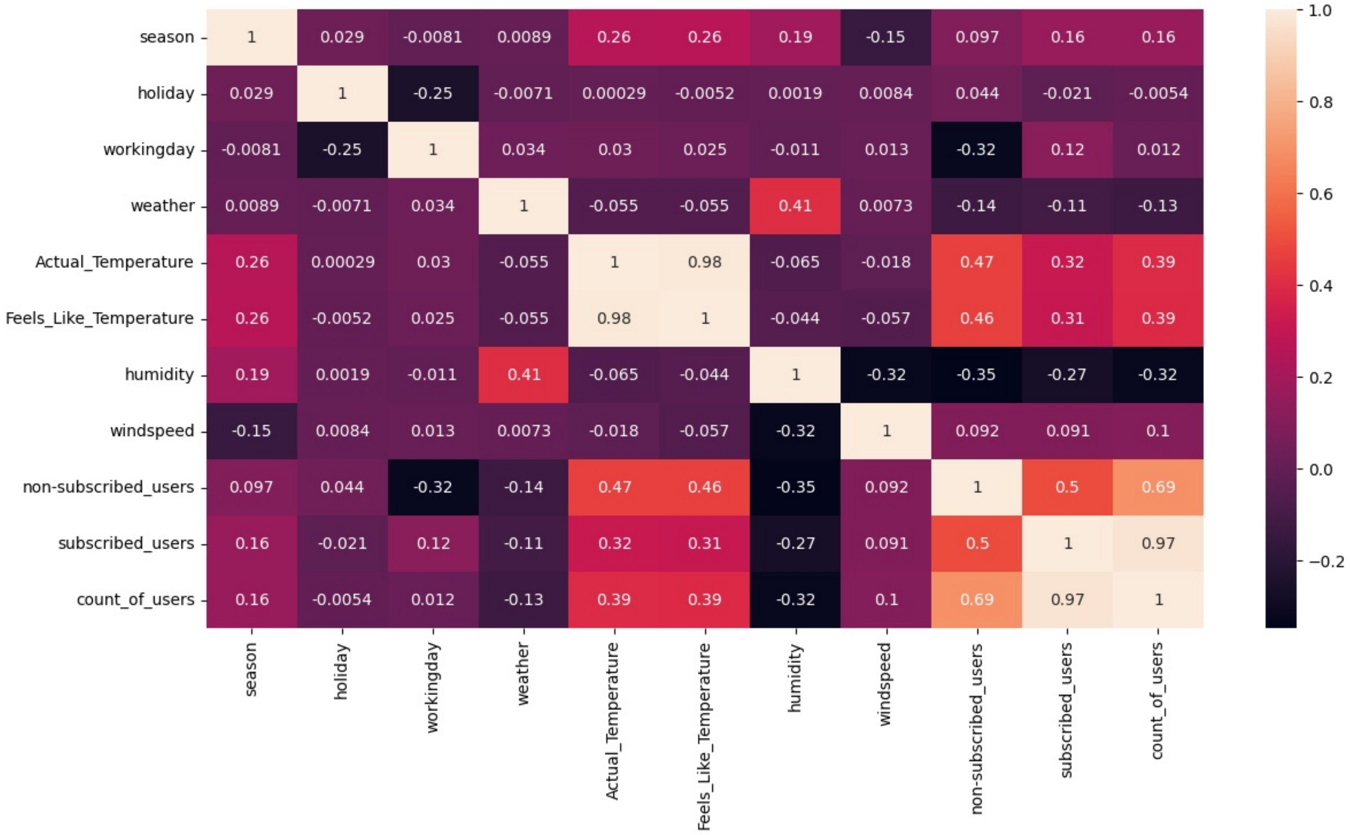
Shape of the Distribution: The distribution of 'count of users' is highly right-skewed (positively skewed). This means that the tail of the distribution extends much further to the right, indicating that there are many instances with a low count of users and fewer instances with a very high count of users.

- **Peak Concentration**: The highest density (most frequent occurrence) of 'count of users' is very close to 0, specifically peaking just above 0. This suggests that the most common scenario is a very low number of users.
- **Rapid Decline**: After the initial peak, the density drops off sharply and then gradually tapers as the 'count of users' increases. This illustrates that as the number of users grows, the frequency of observing that count decreases significantly.
- **Range of Users**: The 'count of users' ranges approximately from 0 up to around 1000, with a very long, thin tail extending towards the higher values.

The KDE plot for 'count of users' reveals a distribution where very low user counts are extremely common, and the frequency of observing higher user counts diminishes rapidly. This is typical for data where many entities have minimal engagement or presence, while only a few have a large following or high activity.

Ckecking Correlation between the features

In [ ]:
```
figure(figsize=(14,     7))
heatmap(df.select_dtypes(include='number').corr(), annot=True)
show()
```



Temperature Variables are Highly Correlated:

- **Actual_Temperature** and **Feels_Like_Temperature** have an extremely strong positive correlation (0.98). This is expected, as "feels like" temperature is typically derived from actual temperature with adjustments for humidity and wind.
- Both **Actual_Temperature** and **Feels_Like_Temperature** show moderate positive correlations with season (0.26 for both). This makes sense as temperature varies with the season.

User Counts and Their Components:

- **count_of_users** is very strongly positively correlated with **subscribed_users** (0.97) and **non-subscribed_users** (0.69). This is also expected, as **count_of_users** is likely the sum or a combination of these two categories.
- **subscribed_users** and **non-subscribed_users** have a moderate positive correlation (0.50), suggesting that periods with more non-subscribed users also tend to have more subscribed users, or vice-versa.

Environmental Factors and User Counts:

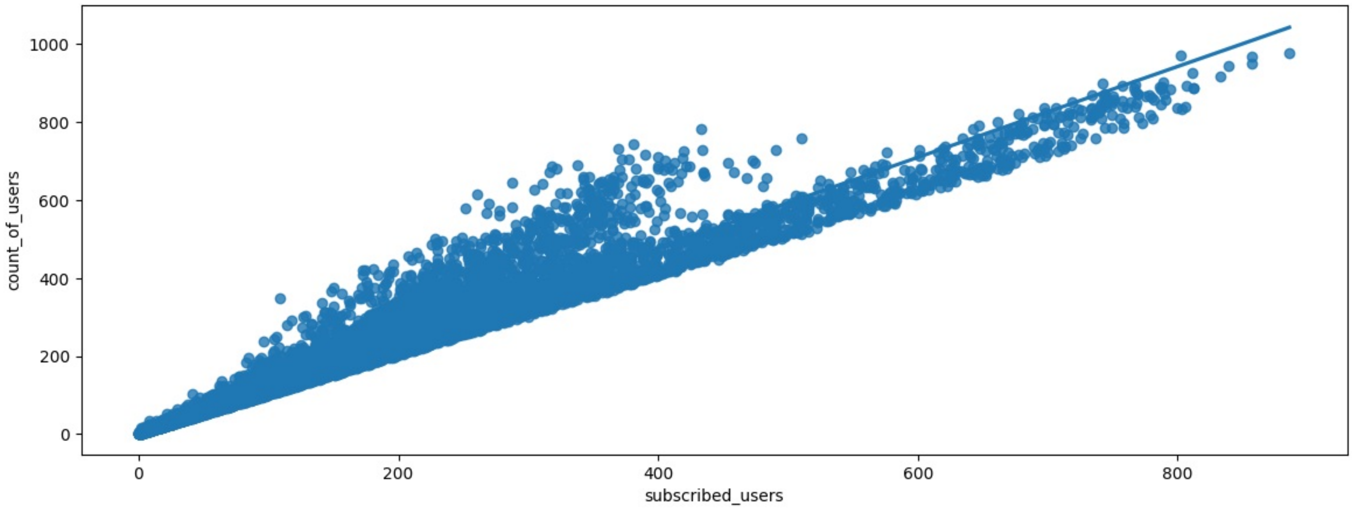
- **Actual_Temperature** and **Feels_Like_Temperature** show moderate positive correlations with **non-subscribed_users** (0.47 and 0.46 respectively) and **subscribed_users** (0.32 and 0.31). This suggests that warmer temperatures might be associated with a higher count of both types of users.
- **humidity** has a moderate negative correlation with **non-subscribed_users** (-0.35) **and subscribed_users** (-0.27), and consequently with **count_of_users** (-0.32). This implies that higher humidity might be associated with fewer users.
- **windspeed** shows very weak correlations with all user count variables (around 0.1 or less), suggesting it's not a strong predictor of user numbers.

Other Notable Correlations:

- weather has a moderate positive correlation with **humidity** (0.41) and a weak negative correlation with **Actual_Temperature** and **Feels_Like_Temperature** (-0.055 for both). This indicates that certain weather conditions are associated with higher **humidity** and slightly lower temperatures.
- **workingday** has a weak negative correlation with **holiday** (-0.25), which is logical as a working day is typically not a holiday.
- **workingday** also has a weak negative correlation with **non-subscribed_users** (-0.32), suggesting fewer non-subscribed users on working days.

There is a high positive correlation (0.97) between **subscribed_users** and **count_of_users**

In [ ]:
```
fig,ax = subplots()
fig.set_size_inches(14, 5)
regplot(x="subscribed_users", y="count_of_users", data=df,ax=ax)
show()
```



Feature Extraction

Extracting new column **[date]** from **[datetime]**

In [ ]:
```
"2011-01-01 00:00:00".split()
```

Out[ ]: `['2011-01-01', '00:00:00']`

In [ ]:
```
df['datetime'][0].split()
```

Out[ ]: `['2011-01-01', '00:00:00']`

In [ ]:
```
df['datetime'][0].split()[0]
```

Out[ ]: `'2011-01-01'`

In [ ]:
```
df["date"] = df['datetime'].apply(lambda x : x.split()[0])
```

In [ ]:
```
df['date'].head()
```

Out[ ]:
```
0    2011-01-01
1    2011-01-01
2    2011-01-01
3    2011-01-01
4    2011-01-01
Name: date, dtype: object
```

Extracting new column **[time]** from **[dateime]**

In [ ]:
```
df['datetime'][0].split()
```

Out[ ]: `['2011-01-01', '00:00:00']`

In [ ]:
```
df['datetime'][0].split()[1]
```

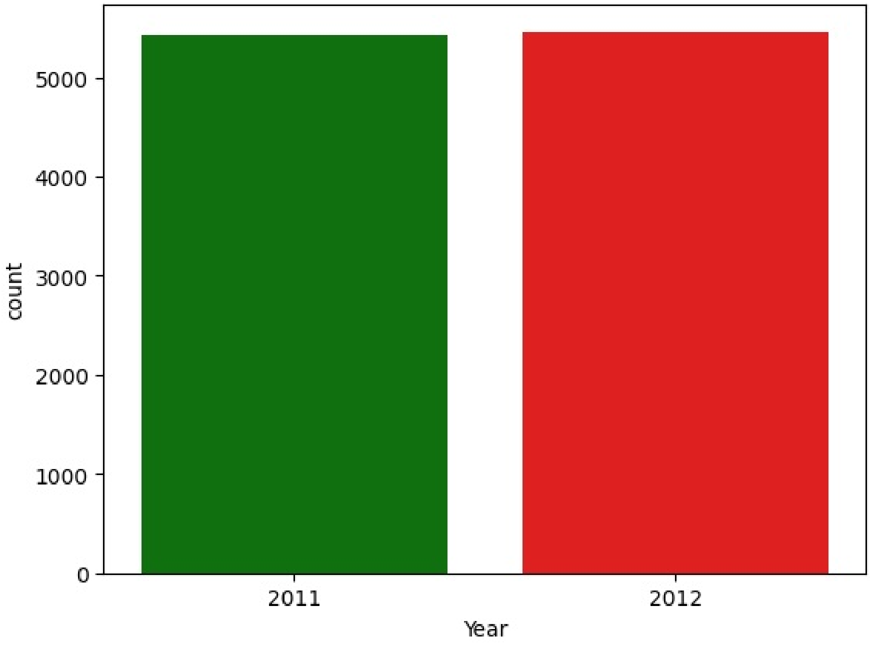
Out[ ]: `'00:00:00'`

In [ ]:
```
df['time'] = df['datetime'].apply(lambda x : x.split()[1])
```

In [ ]:
```
df['time'].head()
```

Out[ ]:
```
0    00:00:00
1    01:00:00
2    02:00:00
3    03:00:00
4    04:00:00
Name: time, dtype: object
```

Extracting new column **[year]** from **[date]**

In [ ]:
```
df['date'][0].split()[0]
```
Out[ ]:
```
'2011-01-01'
```
In [ ]:
```
df['date'][0].split()[0].split('-')[0]
```
Out[ ]:
```
'2011'
```
In [ ]:
```
df['year'] =df['date'].apply(lambda x : x.split()[0].split('-')[0])
```
In [ ]:
```
df['year'].value_counts()
```
Out[ ]:
```
year
2012    5464
2011    5422
Name: count, dtype: int64
```
In [ ]:
```
countplot(x='year', data=df, palette=['green','red'])
xlabel('Year')
show()
```
```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\3002803647.py:1: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable
to `hue` and set `legend=False` for the same effect.

  countplot(x='year', data=df, palette=['green','red'])
```


Extracting new column **[month]** from **[date]**

In [ ]:
```
df['date'][0].split()[0]
```
Out[ ]:
```
'2011-01-01'
```
In [ ]:
```
df['date'][0].split()[0].split('-')[1]
```
Out[ ]:
```
'01'
```
In [ ]:
```
df['date'].apply(lambda x : calendar.month_name[datetime.strptime(x,"%Y-%m-%d").month])
```
Out[ ]:
```
0         January
1         January
2         January
3         January
4         January
           ...
10881    December
10882    December
10883    December
10884    December
10885    December
Name: date, Length: 10886, dtype: object
```

In [ ]:
```
df["month"] = df['date'].apply(lambda x : calendar.month_name[datetime.strptime(x,"%Y-%m-%d").month])
```

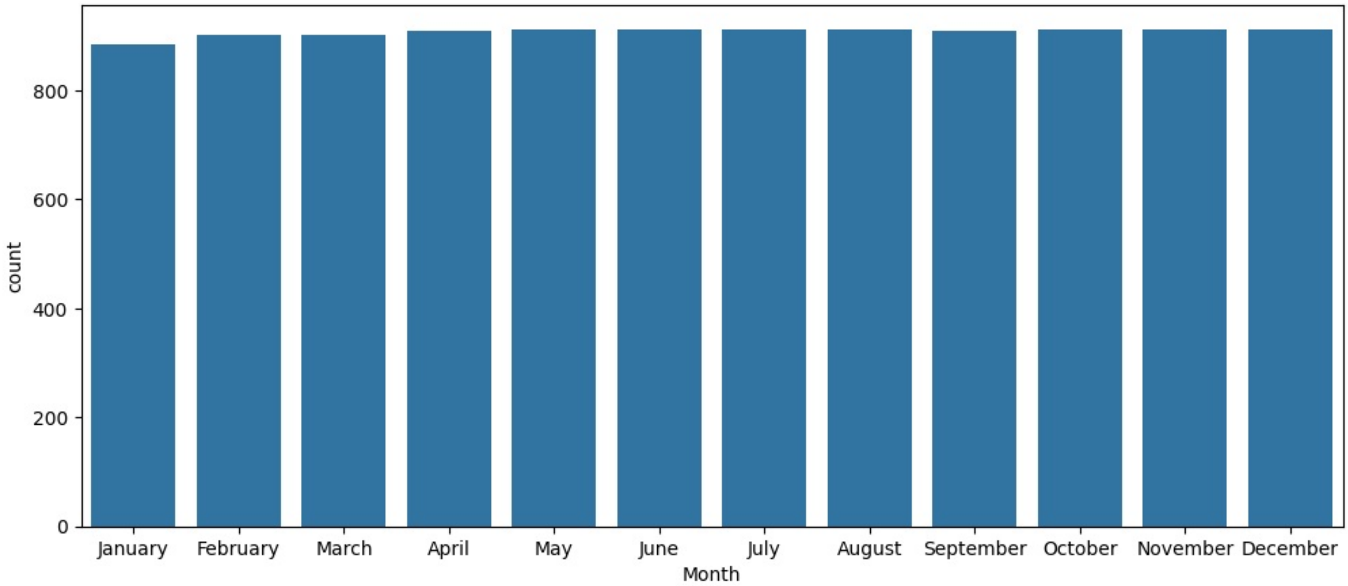
In [ ]:
```
df['month'].head()
```

Out[ ]:
```
0    January
1    January
2    January
3    January
4    January
Name: month, dtype: object
```

In [ ]:
```
df['month'].value_counts()
```

Out[ ]:
```
month
May          912
June         912
July         912
August       912
December     912
October      911
November     911
April        909
September    909
February     901
March        901
January      884
Name: count, dtype: int64
```

In [ ]:
```
fig,ax= subplots()
fig.set_size_inches(12,5)
countplot(x='month', data=df)
xlabel('Month')
show()
```



Extracting new column **[day]** from **[date]**

In [ ]:
```
df['date'][0].split()[0]
```

Out[ ]:
```
'2011-01-01'
```

In [ ]:
```
df['date'][0].split()[0].split('-')[2]
```

Out[ ]:
```
'01'
```

In [ ]:
```
df['date'].apply(lambda x : calendar.day_name[datetime.strptime(x,"%Y-%m-%d").weekday()])
```

```
Out[ ]: 0        Saturday
        1        Saturday
        2        Saturday
        3        Saturday
        4        Saturday
                   ...
        10881    Wednesday
        10882    Wednesday
        10883    Wednesday
        10884    Wednesday
        10885    Wednesday
        Name: date, Length: 10886, dtype: object
```

```
In [ ]: df["day"] = df['date'].apply(lambda x : calendar.day_name[datetime.strptime(x,"%Y-%m-%d").weekday()])
```

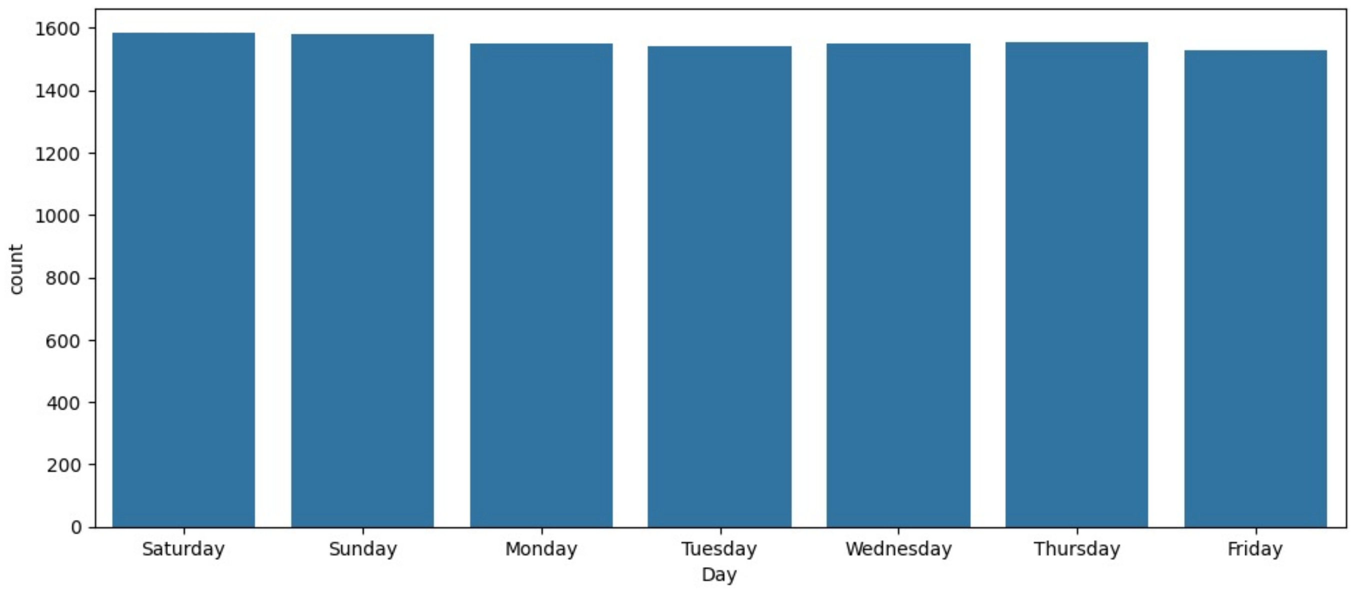
```
In [ ]: df["day"].head()
```

```
Out[ ]: 0    Saturday
        1    Saturday
        2    Saturday
        3    Saturday
        4    Saturday
        Name: day, dtype: object
```

```
In [ ]: df["day"].value_counts()
```

```
Out[ ]: day
        Saturday     1584
        Sunday       1579
        Thursday     1553
        Monday       1551
        Wednesday    1551
        Tuesday      1539
        Friday       1529
        Name: count, dtype: int64
```

```
In [ ]: fig,ax= subplots()
        fig.set_size_inches(12,5)
        countplot(x='day', data=df)
        xlabel('Day')
        show()
```



Extracting new column **[weekend]** from **[day]**

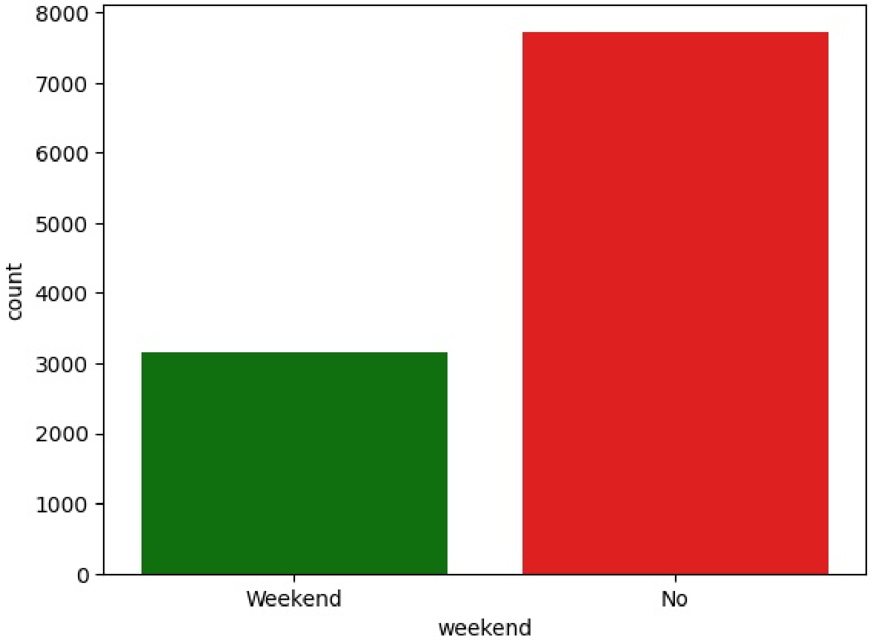
```
In [ ]: def WeekEnd(day):
            day = str(day)
            if day == 'Saturday' or day == 'Sunday':
                return 'Weekend'
            else:
                return 'No'

        df['weekend'] = df['day'].apply(lambda x : WeekEnd(x))
```

```
In [ ]: df['weekend'].value_counts()
```

Out[ ]:
```
weekend
No         7723
Weekend    3163
Name: count, dtype: int64
```

In [ ]:
```
countplot(x='weekend', data=df, palette=['green','red'])
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\565876869.py:1: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable
to `hue` and set `legend=False` for the same effect.

  countplot(x='weekend', data=df, palette=['green','red'])
```



Extracting new column **[hour]** from **[time]**

In [ ]:
```
df['time'][0]
```

Out[ ]:
```
'00:00:00'
```

In [ ]:
```
df['time'][0].split(':')[0]
```

Out[ ]:
```
'00'
```

In [ ]:
```
df['time'].apply(lambda x : x.split(":")[0])
```

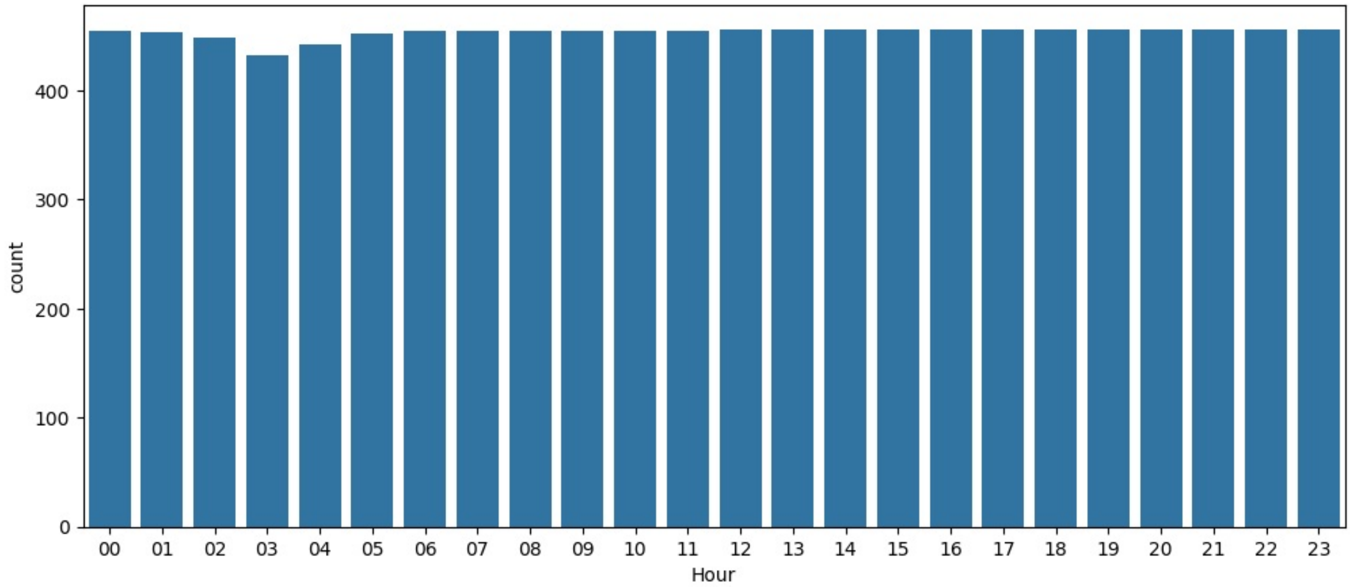
Out[ ]:
```
0        00
1        01
2        02
3        03
4        04
         ..
10881    19
10882    20
10883    21
10884    22
10885    23
Name: time, Length: 10886, dtype: object
```

In [ ]:
```
df["hour"] = df['time'].apply(lambda x : x.split(":")[0])
```

In [ ]:
```
df['hour'].value_counts()
```

Out[ ]:
```
hour
12    456
13    456
22    456
21    456
20    456
19    456
18    456
17    456
16    456
15    456
14    456
23    456
11    455
10    455
09    455
08    455
07    455
06    455
00    455
01    454
05    452
02    448
04    442
03    433
Name: count, dtype: int64
```

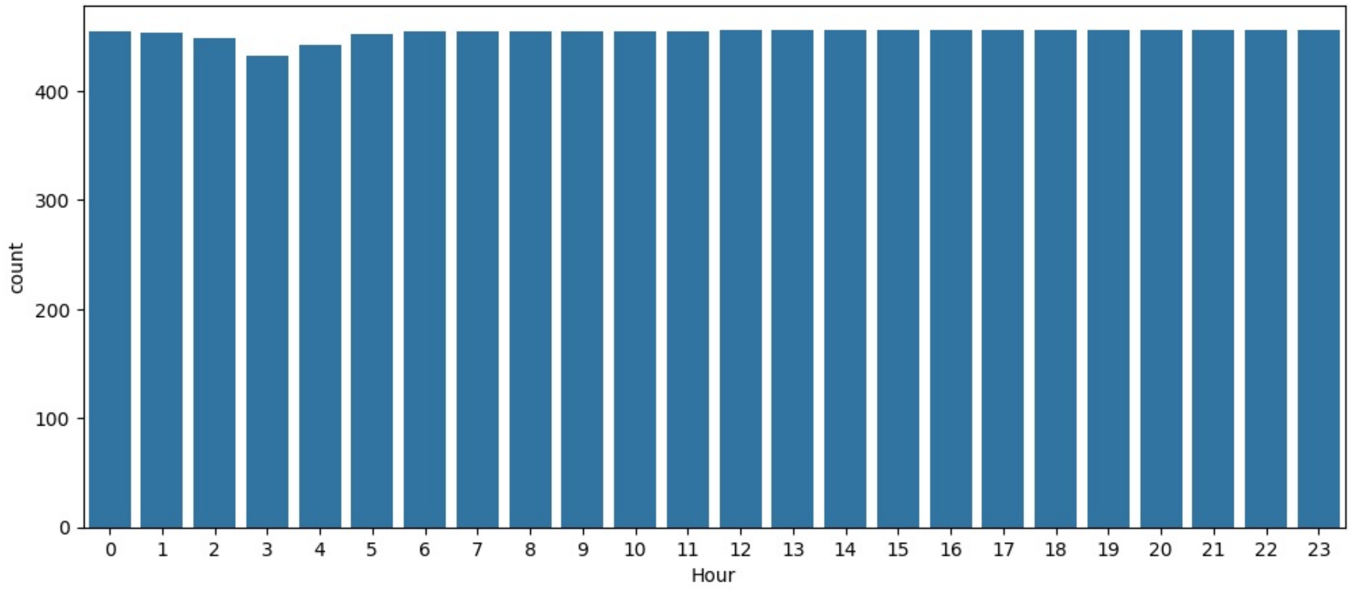
In [ ]:
```
fig,ax= subplots()
fig.set_size_inches(12,5)
countplot(x='hour', data=df)
xlabel('Hour')
show()
```



Extracting new column **[the_usual_time_periods_per_day]** from **[hour]**

In [ ]:
```
df['hour'] = df['hour'].astype(int)
```

In [ ]:
```
fig,ax= subplots()
fig.set_size_inches(12,5)
countplot(x='hour', data=df)
xlabel('Hour')
show()
```

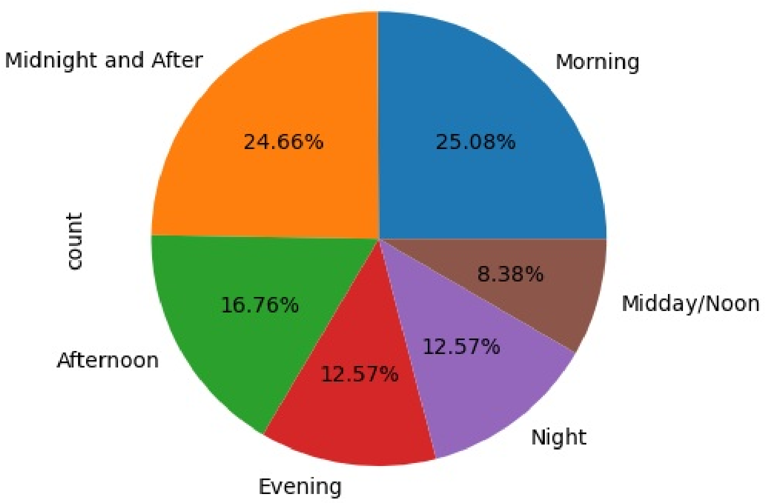


In [ ]:
```
def TheUsualTimePeriodsPerDay(hour):
    hour = int(hour)
    if 6 <= hour <= 11:
        return 'Morning'
    elif 12 <= hour <= 13:
        return 'Midday/Noon'
    elif 14 <= hour <= 17:
        return 'Afternoon'
    elif 18 <= hour <= 20:
        return 'Evening'
    elif 21 <= hour <= 23:
        return 'Night'
    else: # 0 <= hour <= 5
        return 'Midnight and After'

df['the_usual_time_periods_per_day'] = df['hour'].apply(lambda x : TheUsualTimePeriodsPerDay(x))
```

In [ ]:
```
df['the_usual_time_periods_per_day'].value_counts()
```

Out[ ]:
```
the_usual_time_periods_per_day
Morning               2730
Midnight and After    2684
Afternoon             1824
Evening               1368
Night                 1368
Midday/Noon            912
Name: count, dtype: int64
```

In [ ]:
```
df['the_usual_time_periods_per_day'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



Extracting new column **[weather_feeling]** from **[Feels_Like_Temperature]**

In [ ]:
```
df['Feels_Like_Temperature'].unique()
```

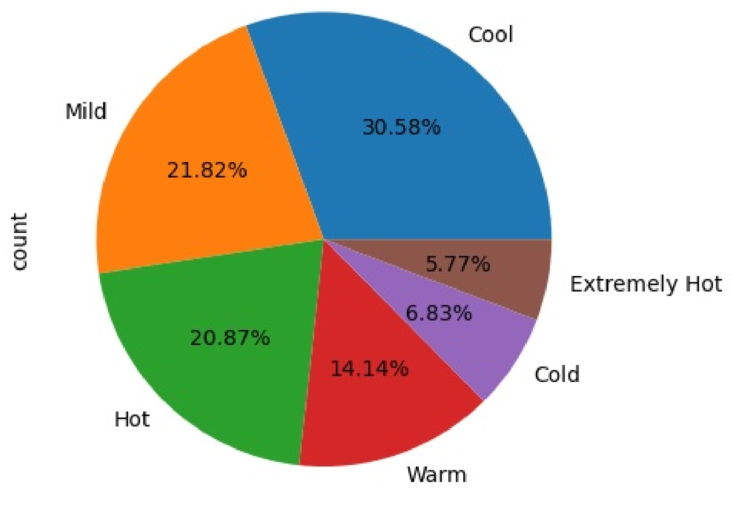
Out[ ]:
```
array([14.395, 13.635, 12.88 , 17.425, 19.695, 16.665, 21.21 , 22.725,
       21.97 , 20.455, 11.365, 10.605,  9.85 ,  8.335,  6.82 ,  5.305,
        6.06 ,  9.09 , 12.12 ,  7.575, 15.91 ,  3.03 ,  3.79 ,  4.545,
       15.15 , 18.18 , 25.   , 26.515, 27.275, 29.545, 23.485, 25.76 ,
       31.06 , 30.305, 24.24 , 18.94 , 31.82 , 32.575, 33.335, 28.79 ,
       34.85 , 35.605, 37.12 , 40.15 , 41.665, 40.91 , 39.395, 34.09 ,
       28.03 , 36.365, 37.88 , 42.425, 43.94 , 38.635,  1.515,  0.76 ,
        2.275, 43.18 , 44.695, 45.455])
```

In [ ]:
```
def classify_temperature(FeelsLikeTemperature):

    if FeelsLikeTemperature < 0:
        return 'Freezing Cold'
    elif FeelsLikeTemperature < 11:
        return 'Cold'
    elif FeelsLikeTemperature < 21:
        return 'Cool'
    elif FeelsLikeTemperature < 26:
        return 'Mild'
    elif FeelsLikeTemperature < 31:
        return 'Warm'
    elif FeelsLikeTemperature < 36:
        return 'Hot'
    else:
        return 'Extremely Hot'

df['weather_feeling'] = df['Feels_Like_Temperature'].apply(lambda x : classify_temperature(x))
```

In [ ]:
```
df['weather_feeling'].value_counts()
```

Out[ ]:
```
weather_feeling
Cool             3329
Mild             2375
Hot              2272
Warm             1539
Cold              743
Extremely Hot     628
Name: count, dtype: int64
```

In [ ]:
```
df['weather_feeling'].value_counts().plot.pie(autopct='%0.2f%%')
show()
```



Mapping **season column**

In [ ]:
```
df['season'].value_counts()
```

Out[ ]:
```
season
4    2734
2    2733
3    2733
1    2686
Name: count, dtype: int64
```

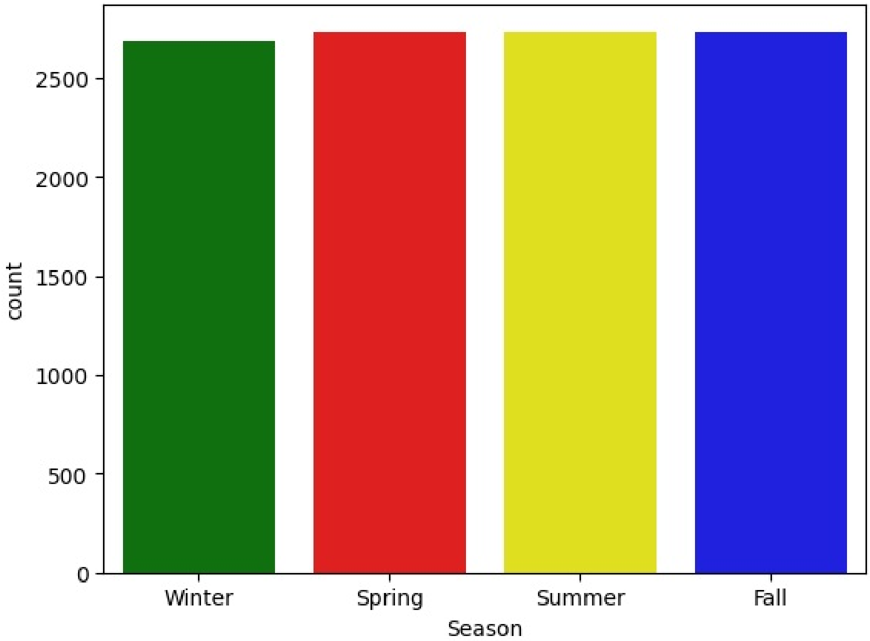
In [ ]:
```
dictionnaire_saisons = {1 :"Winter" , 2: "Spring", 3 : "Summer", 4 : "Fall"}
df["season"] = df["season"].map(dictionnaire_saisons)
```

In [ ]:
```
df['season'].value_counts()
```

Out[ ]:
```
season
Fall      2734
Spring    2733
Summer    2733
Winter    2686
Name: count, dtype: int64
```

In [ ]:
```
countplot(x='season', data=df, palette=['green','red','yellow','blue'])
xlabel('Season')
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\2074212223.py:1: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable
to `hue` and set `legend=False` for the same effect.

  countplot(x='season', data=df, palette=['green','red','yellow','blue'])
```



Mapping **weather column**

Clear + Few clouds + Partly cloudy + Partly cloudy ----> **Clear to Partly Cloudy**

Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist ----> **Mist and Cloudy**

Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds" ----> **Light Precipitation**

Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog ----> **Severe Weather**

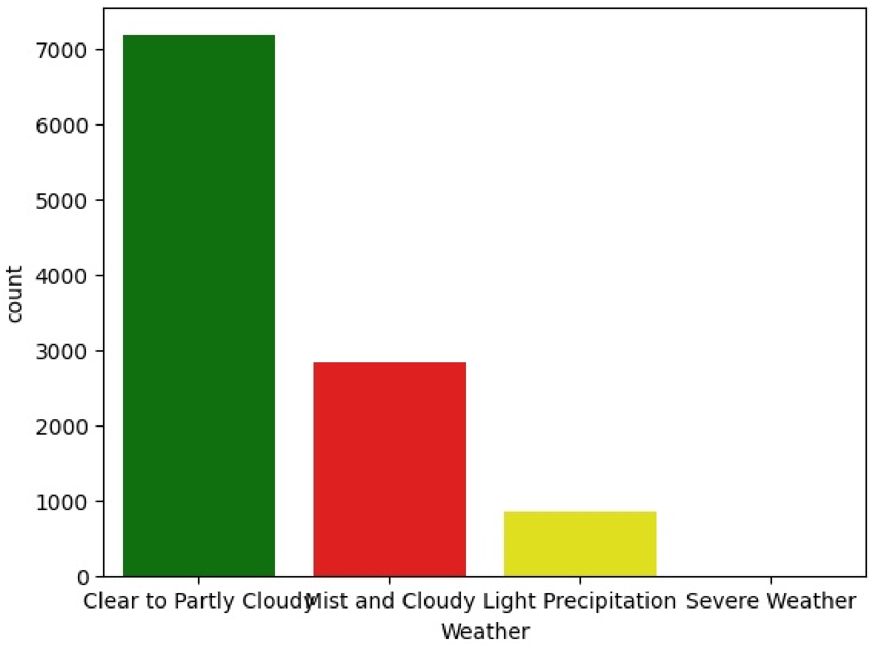
In [ ]:
```
df["weather"].value_counts()
```

Out[ ]:
```
weather
1    7192
2    2834
3     859
4       1
Name: count, dtype: int64
```

In [ ]:
```
df["weather"] = df["weather"].map({1: "Clear to Partly Cloudy",\
                                   2 : "Mist and Cloudy", \
                                   3 : "Light Precipitation", \
                                   4 :"Severe Weather" })
```

In [ ]:
```
countplot(x='weather', data=df, palette=['green','red','yellow','blue'])
xlabel('Weather')
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\3207305020.py:1: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable
to `hue` and set `legend=False` for the same effect.

  countplot(x='weather', data=df, palette=['green','red','yellow','blue'])
```



Detect Outliers

In [ ]:

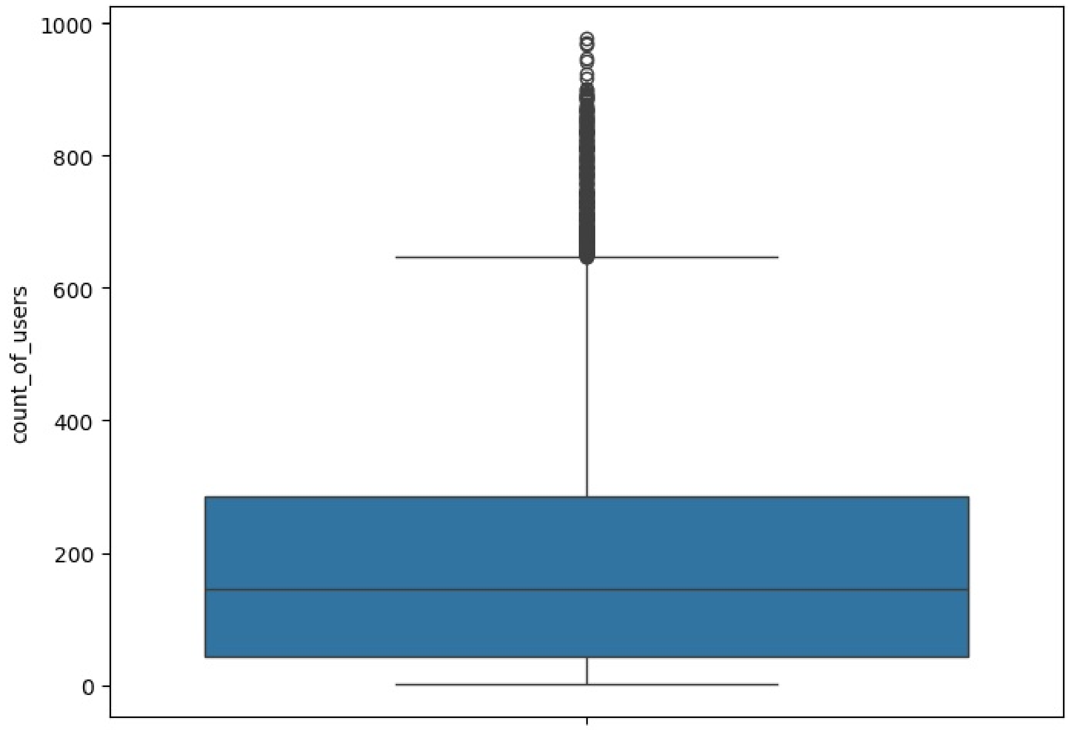
```
df[['count_of_users']].describe()
```

Out[ ]:

	count_of_users
count	10886.000000
mean	191.574132
std	181.144454
min	1.000000
25%	42.000000
50%	145.000000
75%	284.000000
max	977.000000

In [ ]:

```
fig, axes = subplots(nrows=1,ncols=1)
fig.set_size_inches(8, 6)
boxplot(data=df,y="count_of_users",orient="v",ax=axes)
show()
```

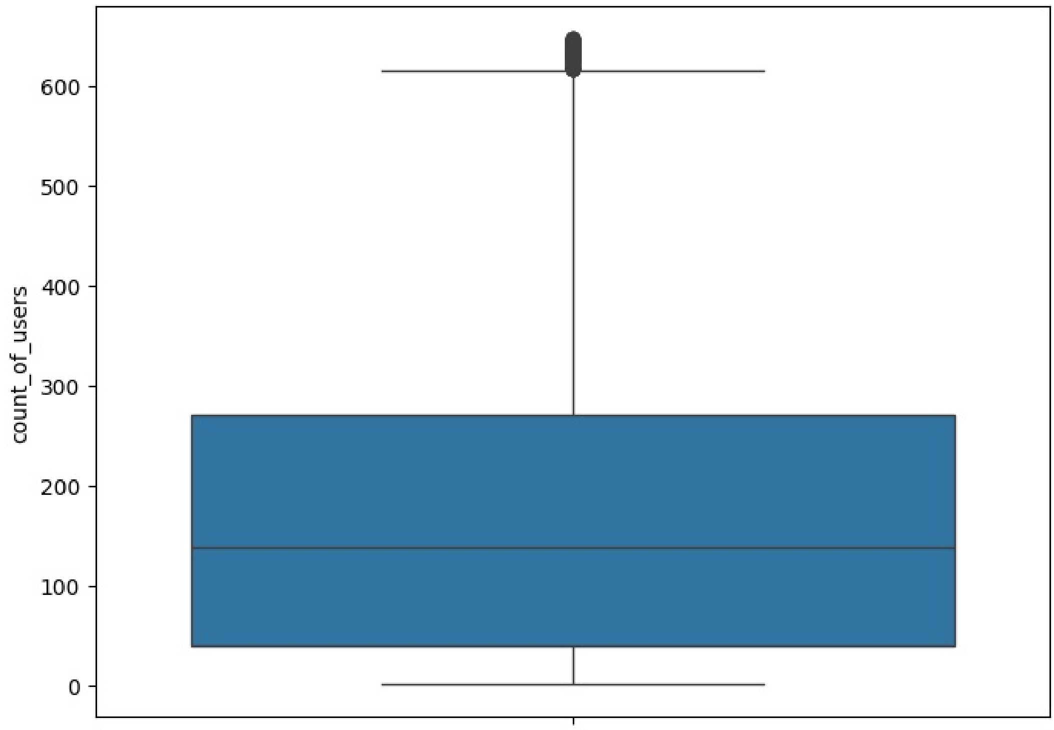


- **Median**: The median 'count_of_users' is relatively low, appearing to be around 150-200. This means that at least half of the observations have a user count below this value.
- **Spread of Middle 50%**: The box itself is quite tall, indicating a considerable spread in the middle 50% of the 'count_of_users'. The range from Q1 to Q3 is relatively wide.
- **Skewness**: The box plot shows a clear right-skewness.
- The median line is closer to the bottom of the box, and the upper whisker is significantly longer than the lower one (though the lower whisker is very short, almost at 0).
- This indicates that there are more data points with lower user counts, and a long tail of higher user counts.
- **Outliers**: There are numerous data points plotted as individual circles above the upper whisker. These are outliers, representing instances with significantly higher 'count_of_users' than the typical range. The highest outlier is close to 1000 users.
- **Lower Bound**: The lower whisker extends down to approximately 0, suggesting that user counts can be very low.
- The majority of observations have a relatively low number of users.
- The data is heavily skewed towards lower user counts, meaning high user counts are less frequent.
- There's a significant presence of outliers, indicating that while most instances have low to moderate user counts, there are a good number of instances with exceptionally high user counts.

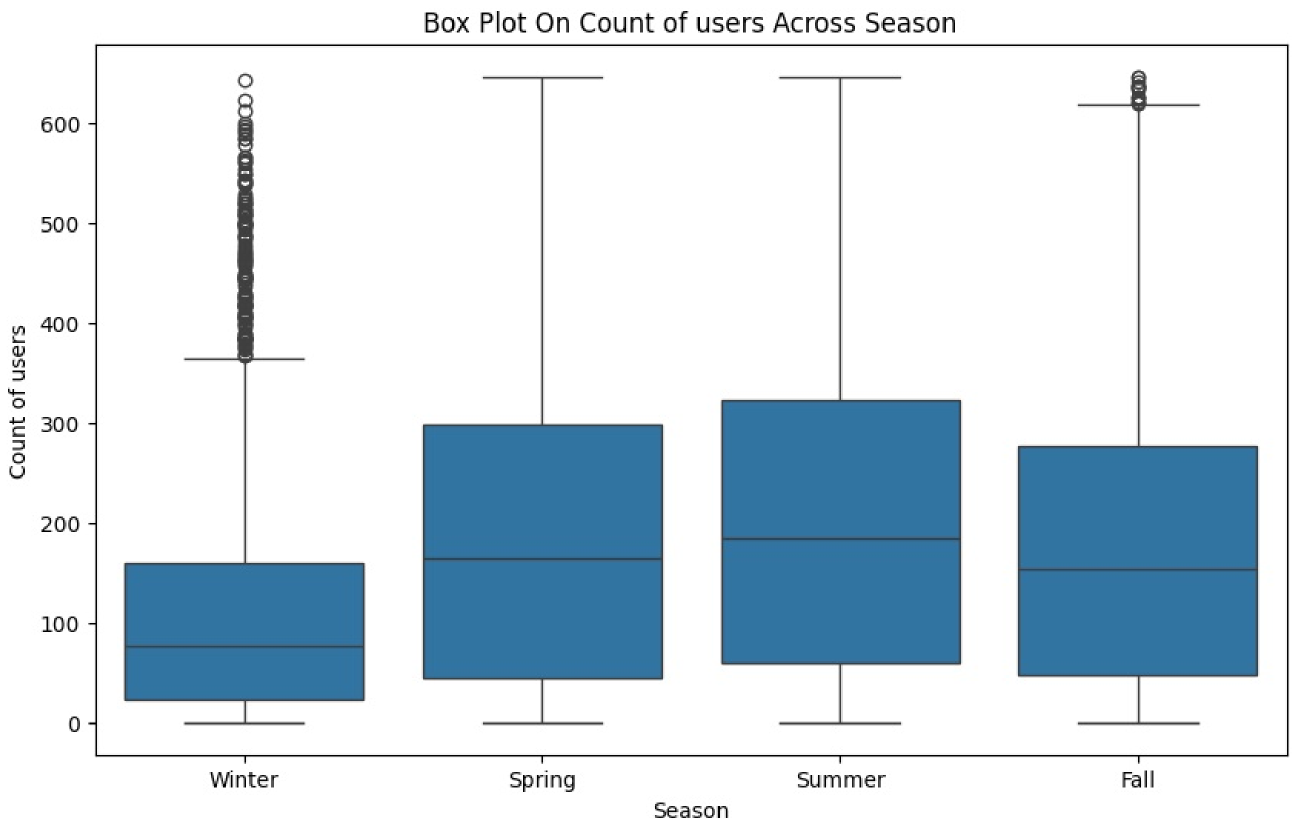
In [ ]:
```
numerical_data=df[['count_of_users']]
for column in numerical_data.columns:
    Q1=numerical_data[column].quantile(0.25)
    Q3=numerical_data[column].quantile(0.75)
    IQR = Q3-Q1

    Lower_bound = Q1 - 1.5*IQR
    Upper_bound = Q3 + 1.5*IQR

    outliers = ((numerical_data[column]>Upper_bound)|(numerical_data[column]<Lower_bound)).sum()
    Total = numerical_data[column].shape[0]
    print(f'Total of outliers in {column} are   :   {outliers}--{round(100*(outliers)/Total,2)}%')

    if outliers > 0:
        df=df.loc[(df[column] <= Upper_bound) & (df[column] >= Lower_bound)]
```

```
Total of outliers in count_of_users are   :   300--2.76%
```

In [ ]:
```
fig, axes = subplots(nrows=1,ncols=1)
fig.set_size_inches(8, 6)
boxplot(data=df,y="count_of_users",orient="v",ax=axes)
show()
```




```
In [ ]: fig, axes = subplots(nrows=1,ncols=1)
        fig.set_size_inches(10, 6)
        boxplot(data=df,y="count_of_users",x="season",orient="v",ax=axes)
        axes.set(xlabel='Season', ylabel='Count of users',title="Box Plot On Count of users Across Season")
        show()
```



- **Outliers**: All seasons show a significant number of outliers, particularly on the higher end of the 'count_of_users' scale. This indicates that regardless of the season, there are instances where user counts are exceptionally high.
- **Skewness**: All distributions appear to be right-skewed, with the median closer to the bottom of the box and longer upper whiskers/outlier ranges. This suggests that in every season, lower user counts are more common, with fewer instances of very high user counts.

Season-Specific Observations:

Winter:

- Has the lowest median 'count_of_users' compared to other seasons (around 70-80).
- The interquartile range (IQR) is relatively narrow, suggesting less variability in user counts during winter compared to other seasons.

- Despite the lower typical counts, there are still many high outliers, some reaching over 600 users.

Spring:

- Shows a noticeable increase in median 'count_of_users' compared to Winter (around 170-180).
- The IQR is wider than Winter's, indicating more variability in user counts.
- The upper whisker extends significantly higher, and there are high outliers, though they don't reach as high as some in Winter.

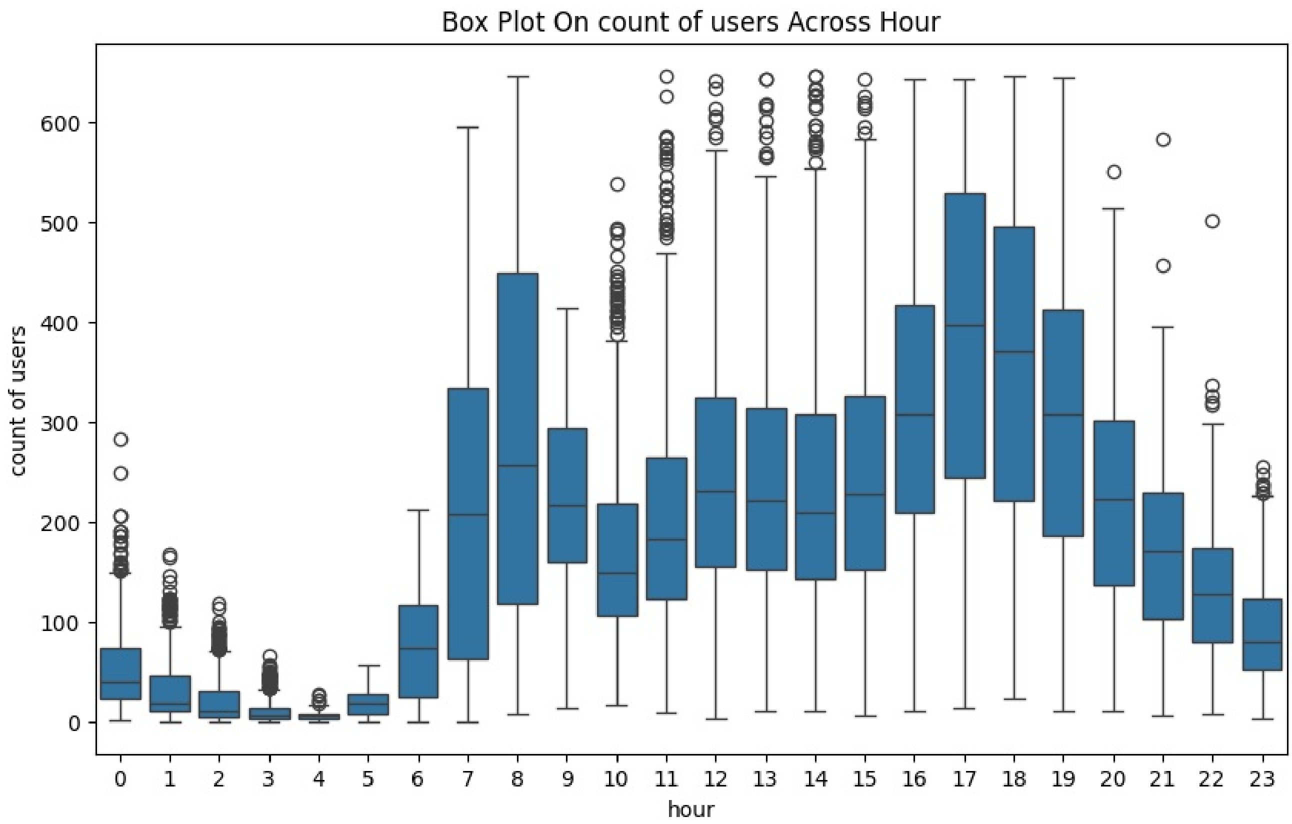
Summer:

- Has the highest median 'count_of_users' among all seasons (around 180-190), very similar to Spring's median.
- The IQR is also wide, comparable to Spring, suggesting similar variability.
- The upper whisker and outliers extend to high values, similar to Spring.

Fall:

- The median 'count_of_users' is slightly lower than Spring and Summer (around 150-160) but still higher than Winter.
- The IQR is relatively wide, similar to Spring and Summer.
- Like other seasons, Fall also exhibits many high outliers, reaching similar maximum values as Winter.

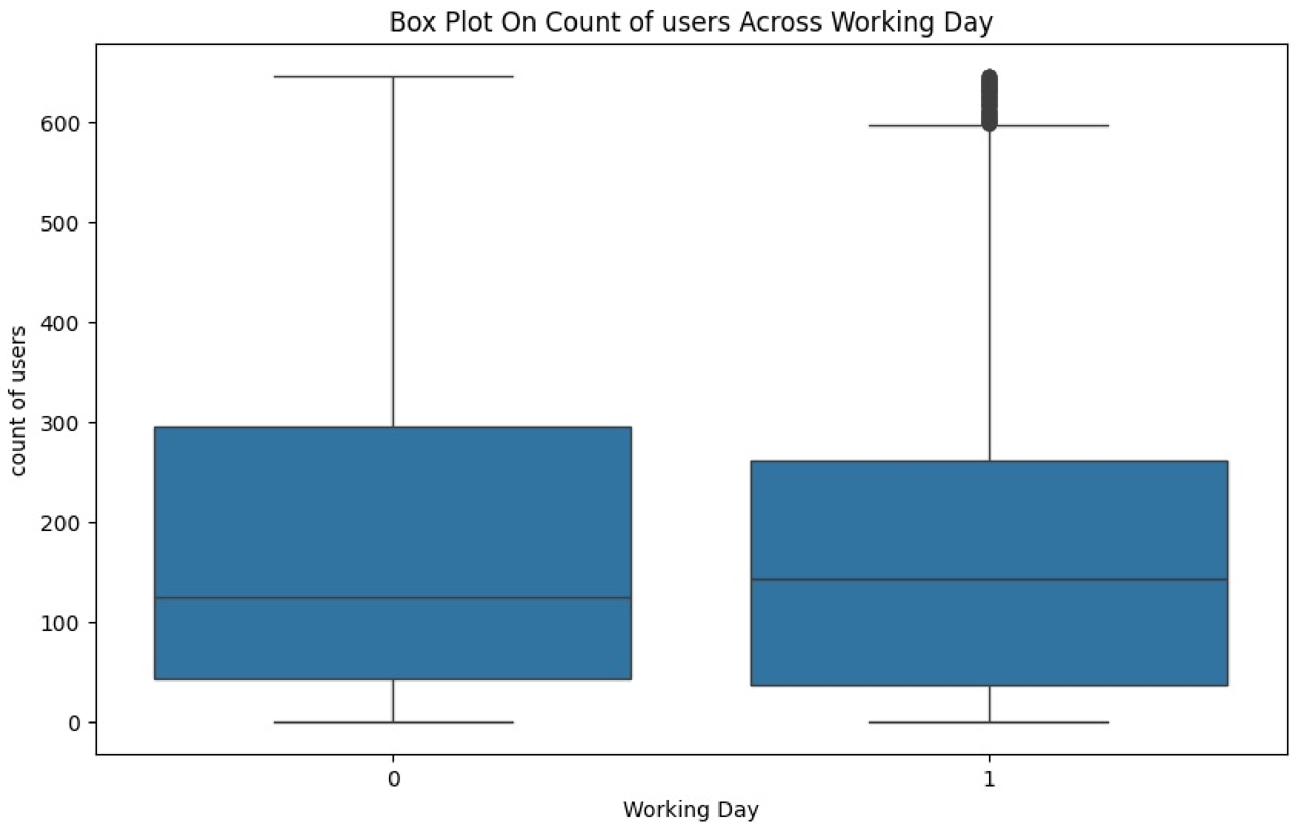
In [ ]:
```
fig, axes = subplots(nrows=1,ncols=1)
fig.set_size_inches(10, 6)
boxplot(data=df,y="count_of_users",x="hour",orient="v",ax=axes)
axes.set(xlabel='hour', ylabel='count of users',title="Box Plot On count of users Across Hour")
show()
```



- **Hourly Variation**: There's a very clear and significant variation in user counts throughout the day.
- **Outliers**: Most hours, especially during peak times, show a considerable number of outliers on the higher end of the 'count_of_users' scale. This indicates that even within typical activity periods, there are instances of exceptionally high user engagement.
- **Skewness**: Most hourly distributions are right-skewed, meaning lower user counts are more common, with a tail extending to higher counts.
- **Hourly Patterns**:
- Early Morning (Hours 0-5):
- User counts are generally very low.
- The median count_of_users is at its lowest, often close to zero.
- The interquartile range (IQR) is very narrow, indicating little variability.

- A few outliers exist, but they are much lower than peak hours.
- Hour 4 and 5 show a slight increase in median and IQR, perhaps indicating the start of the day for some users.
- **Morning/Commute (Hours 6-9)**:
- A sharp increase in user activity begins around Hour 6.
- The median count_of_users rises significantly, with Hour 7 and 8 showing substantial growth.
- Hour 8 and 9 show high median values and a wider IQR, indicating more variability as activity ramps up. Hour 8 has a particularly high median and upper quartile.
- **Daytime (Hours 10-15)**:
- User counts remain high, but the median fluctuates.
- Hour 11, 12, 13, 14, and 15 generally maintain high median user counts, often above 200.
- The IQR remains wide, and many high outliers are present, some reaching over 600 users.
- **Evening/Peak Hours (Hours 16-19)**:
- These hours represent the peak activity period.
- Hour 17 and 18 show the highest median count_of_users (around 500 and 490 respectively), indicating the busiest times of the day.
- The boxes are very tall, showing significant variability in user counts, and there are numerous high outliers.
- Hour 16 and 19 also exhibit high activity, though slightly lower than 17 and 18.
- Late Night (Hours 20-23):
- User counts begin to decline steadily.
- The median count_of_users decreases progressively from Hour 20 to 23.
- The IQR also narrows, and while outliers are still present, their maximum values decrease.

In [ ]:
```
fig, axes = subplots(nrows=1,ncols=1)
fig.set_size_inches(10, 6)
boxplot(data=df,y="count_of_users",x="workingday",orient="v",ax=axes)
axes.set(xlabel='Working Day', ylabel='count of users',title="Box Plot On Count of users Across Working Day")
show()
```



- **Outliers**: Both categories (working day and non-working day) show a significant number of outliers, particularly on the higher end of the 'count_of_users' scale. This indicates that even within typical day types, there are instances of exceptionally high user engagement.
- **Skewness**: Both distributions appear to be right-skewed, with the median closer to the bottom of the box and longer upper whiskers/outlier ranges. This suggests that in both working and non-working days, lower user counts are more common, with fewer instances of very high user counts.

Comparison of Working Day (1) vs. Non-Working Day (0):

- **Median count_of_users**:
- Non-Working Day (0): The median count_of_users is around 120-130.
- Working Day (1): The median count_of_users is slightly higher, around 140-150.
- This suggests that, on average, there might be a slightly higher number of users on working days compared to non-working days.
- **Interquartile Range (IQR)**:
- Non-Working Day (0): The box is relatively tall, indicating a wide spread in the middle 50% of user counts.
- Working Day (1): The box is also quite tall, with a similar or slightly narrower spread compared to non-working days.
- Both day types show considerable variability in user counts within their typical ranges.
- **Range of Data (excluding outliers)**:
- The lower whisker for both categories extends close to 0, indicating that very low user counts can occur on any type of day.
- The upper whiskers extend to similar maximum values for both categories, suggesting that the upper range of typical user counts is similar.
- **Outliers**:

Both categories show a similar range of high outliers, reaching up to around 600-650 users. This implies that exceptionally high user activity can occur on both working and non-working days.

In [ ]:
```
# df.to_csv('Bake rental (New data).csv',index=False)
```

Analysis

- **Visualisation of continuous features vs Number of users**

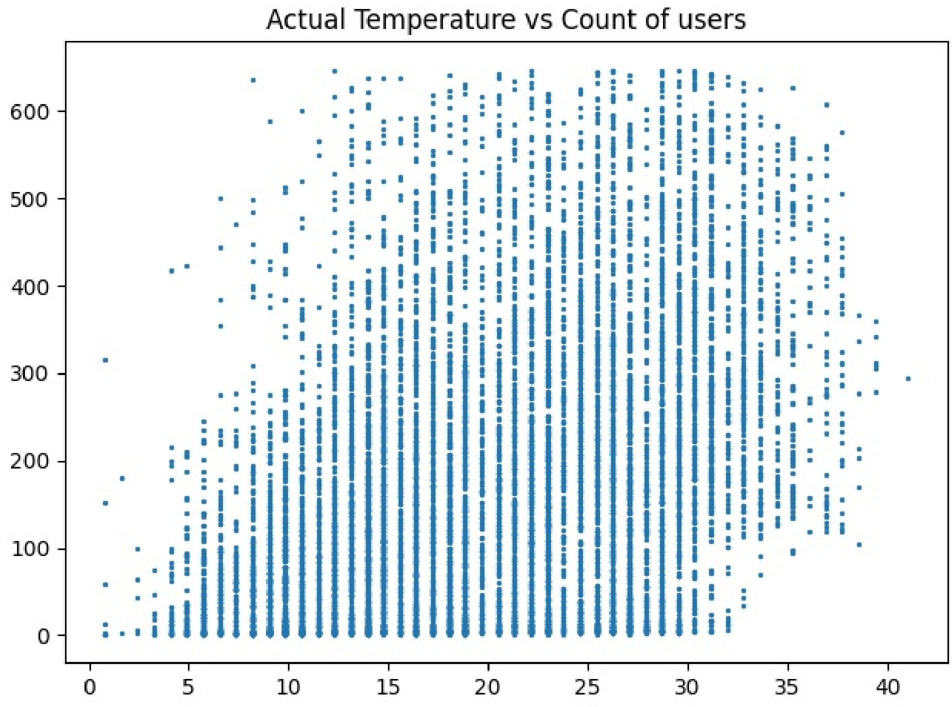
In [ ]:
```
df.head()
```

Out[ ]:

	datetime	season	holiday	workingday	weather	Actual_Temperature	Feels_Like_Temperature	humidity	windspeed	subscribed
0	2011-01-01 00:00:00	Winter	0	0	Clear to Partly Cloudy	9.84	14.395	81	0.0	
1	2011-01-01 01:00:00	Winter	0	0	Clear to Partly Cloudy	9.02	13.635	80	0.0	
2	2011-01-01 02:00:00	Winter	0	0	Clear to Partly Cloudy	9.02	13.635	80	0.0	
3	2011-01-01 03:00:00	Winter	0	0	Clear to Partly Cloudy	9.84	14.395	75	0.0	
4	2011-01-01 04:00:00	Winter	0	0	Clear to Partly Cloudy	9.84	14.395	75	0.0	

5 rows × 21 columns

In [ ]:
```
title('Actual Temperature vs Count of users')
scatter(df['Actual_Temperature'],df['count_of_users'],s=2)
tight_layout()
```

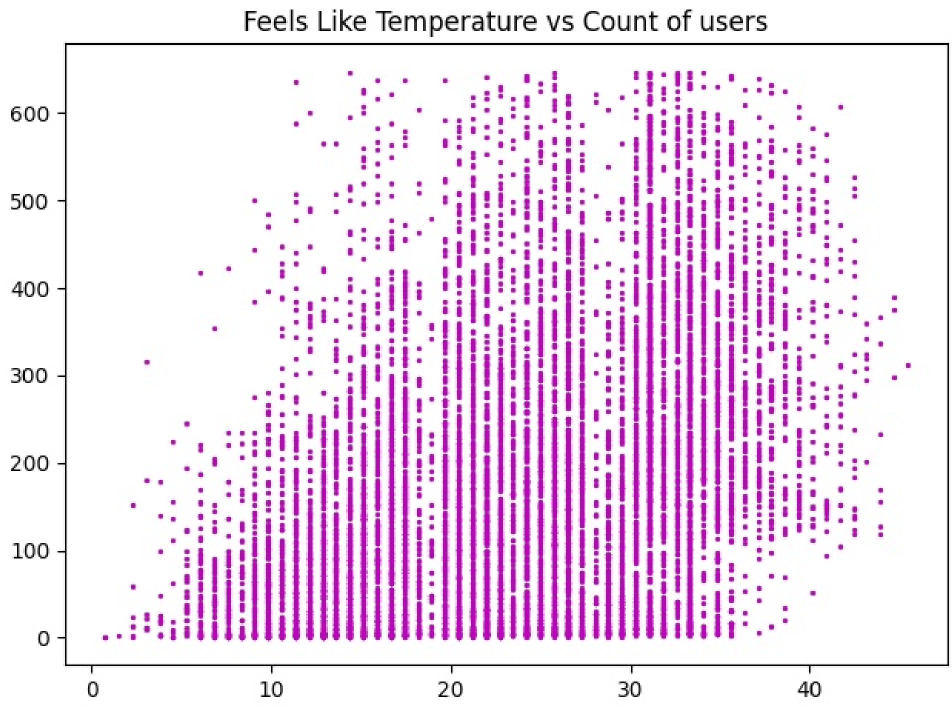


- **Positive Association (up to a point)**: There appears to be a general positive association between 'Actual Temperature' and 'Count of users' for temperatures roughly between 5°C and 25°C. As the temperature increases in this range, the maximum and typical 'Count of users' also tend to increase.
- **Peak Activity Range**: The highest 'Count of users' (reaching above 600) seems to occur within a temperature range of approximately 15°C to 30°C. This suggests an optimal temperature window for user activity.

Decreased Activity at Extremes:

- At very low temperatures (below 5°C), the 'Count of users' is consistently low, mostly below 100.
- At very high temperatures (above 30°C), while there are still some high user counts, the overall density of points and the maximum observed user counts appear to decrease compared to the peak range. The spread of user counts also seems to narrow at very high temperatures.
- **Vertical Bands/Density**: The data points form dense vertical bands, especially in the 10°C to 30°C range. This indicates that for a given temperature, there can be a wide range of 'Count of users' values, from very low to very high.
- **No Clear Linear Relationship**: While there's a general trend, the relationship is not strictly linear. It seems to follow a curvilinear pattern, increasing, peaking, and then potentially decreasing or leveling off.

In [ ]:
```
title('Feels Like Temperature vs Count of users')
scatter(df['Feels_Like_Temperature'],df['count_of_users'],s=2,c='m')
tight_layout()
```

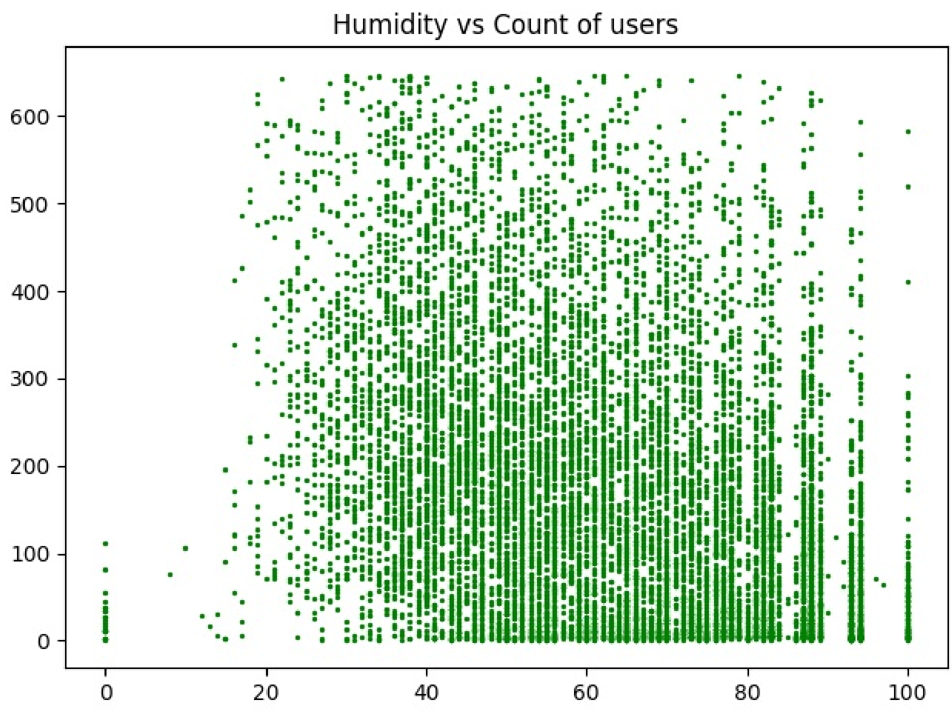


- **Positive Association (up to a point)**: Similar to the 'Actual Temperature' plot, there appears to be a general positive association between 'Feels Like Temperature' and 'Count of users' for temperatures roughly between 5°C and 25°C. As the 'feels like' temperature increases in this range, the maximum and typical 'Count of users' also tend to increase.
- **Peak Activity Range**: The highest 'Count of users' (reaching above 600) seems to occur within a 'feels like' temperature range of approximately 15°C to 35°C. This suggests an optimal 'feels like' temperature window for user activity.

Decreased Activity at Extremes:

- At very low 'feels like' temperatures (below 5°C), the 'Count of users' is consistently low, mostly below 100.
- At very high 'feels like' temperatures (above 35°C), while there are still some high user counts, the overall density of points and the maximum observed user counts appear to decrease compared to the peak range. The spread of user counts also seems to narrow at very high temperatures.
- **Vertical Bands/Density**: The data points form dense vertical bands, especially in the 10°C to 35°C range. This indicates that for a given 'feels like' temperature, there can be a wide range of 'Count of users' values, from very low to very high.
- **No Clear Linear Relationship**: While there's a general trend, the relationship is not strictly linear. It seems to follow a curvilinear pattern, increasing, peaking, and then potentially decreasing or leveling off.

In [ ]:
```
title('Humidity vs Count of users')
scatter(df['humidity'],df['count_of_users'],s=2,c='g')
tight_layout()
```



- **Inverse Relationship (General Trend)**: There appears to be a general inverse or negative relationship between 'Humidity' and 'Count of users' for humidity levels roughly between 20% and 80%. As humidity increases in this range, the maximum and typical 'Count of users' tend to decrease.
- **Peak Activity Range**: The highest 'Count of users' (reaching above 600) seems to occur within a humidity range of approximately 20% to 60%. This suggests that lower to moderate humidity levels are associated with higher user activity.

Decreased Activity at Extremes:

- At very low humidity levels (below 20%), the 'Count of users' is generally low, mostly below 100.
- At very high humidity levels (above 80%), the 'Count of users' also tends to be lower, with fewer instances of high user counts. There's a notable drop-off in the density of points and maximum user counts as humidity approaches 90-100%.
- **Vertical Spread/Density**: For a given humidity level, there can be a wide range of 'Count of users' values, from very low to very high. This is indicated by the dense vertical spread of points across much of the humidity range.

Months groub by mean of count of users

In [ ]:
```
sortOrder = ["January","February","March","April","May","June","July","August","September","October","November"
hueOrder = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]

monthAggregated = DataFrame(df.groupby("month")["count_of_users"].mean()).reset_index()
```

In [ ]:
```
monthAggregated
```

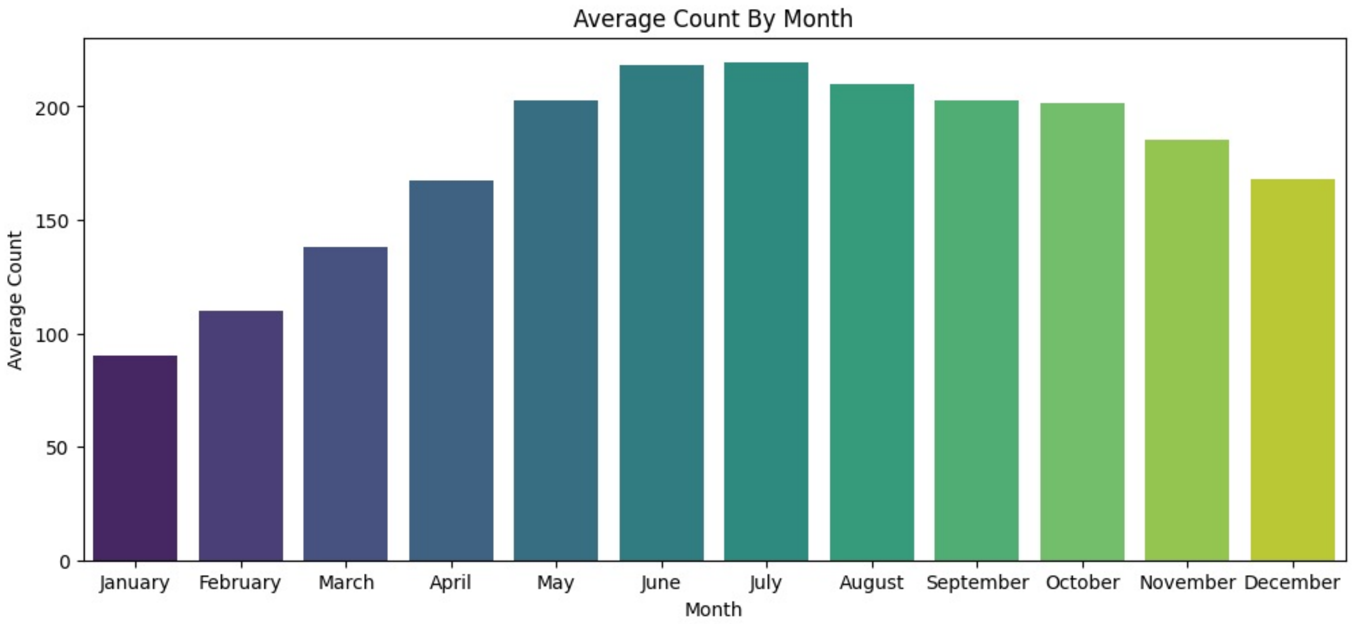
Out[ ]:

	month	count_of_users
0	April	167.282633
1	August	209.696101
2	December	167.947720
3	February	110.003330
4	January	90.366516
5	July	219.409040
6	June	218.017241
7	March	138.040678
8	May	202.437146
9	November	185.039106
10	October	201.269805
11	September	202.606977

In [ ]:

```
fig, ax = subplots()
fig.set_size_inches(12, 5)

barplot(data=monthAggregated, x="month", y="count_of_users", ax=ax, order=sortOrder, palette="viridis")
ax.set(xlabel='Month', ylabel='Average Count', title="Average Count By Month")
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\2799023441.py:4: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  barplot(data=monthAggregated, x="month", y="count_of_users", ax=ax, order=sortOrder, palette="viridis")
```



- **Clear Seasonal Pattern**: There's a very distinct seasonal pattern in the average count.
- **Lowest Counts in Winter**: January has the lowest average count, followed by February.
- **Gradual Increase in Spring**: The average count steadily increases from March through April and May.
- **Peak in Summer**: June and July show the highest average counts, indicating peak activity during these summer months. August also maintains a high average count.
- **Decline in Fall/Winter**: The average count begins to decline from September through October, November, and December, returning to lower levels as winter approaches.
- **Color Gradient**: The bars are colored with a gradient, visually reinforcing the trend from lower counts (darker purple/blue) to higher counts (greens/teals) and back to lower (yellow-green).

Hours and season groub by mean of count of users

In [ ]:

```
hourAggregated = DataFrame(df.groupby(["hour","season"], sort=True)["count_of_users"].mean()).reset_index()
hourAggregated
```

Out[ ]:

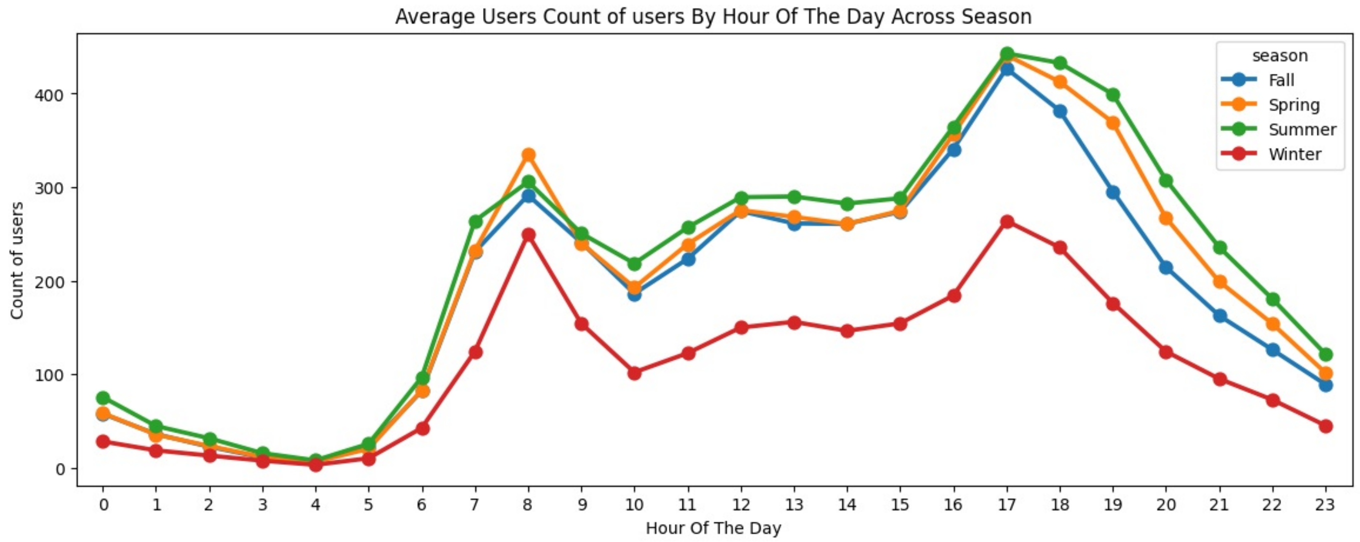
	hour	season	count_of_users
0	0	Fall	57.877193
1	0	Spring	58.473684
2	0	Summer	75.675439
3	0	Winter	28.292035
4	1	Fall	36.166667
...	...	...	...
91	22	Winter	72.912281
92	23	Fall	89.298246
93	23	Spring	101.684211
94	23	Summer	121.719298
95	23	Winter	45.333333

96 rows × 3 columns

In [ ]:

```
fig,ax= subplots()
fig.set_size_inches(14,5)
pointplot(x=hourAggregated["hour"], y=hourAggregated["count_of_users"],hue=hourAggregated["season"], data=hourAg
ax.set(xlabel='Hour Of The Day', ylabel='Count of users',title="Average Users Count of users By Hour Of The Day
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\2794846622.py:3: UserWarning:

The `join` parameter is deprecated and will be removed in v0.15.0.

  pointplot(x=hourAggregated["hour"], y=hourAggregated["count_of_users"],hue=hourAggregated["season"], data=hour
Aggregated, join=True,ax=ax)
```



- **Clear Daily and Seasonal Patterns:** The plot clearly shows distinct daily cycles of user activity, with significant differences between seasons.
- **Two Peaks in User Activity**: For most seasons, there appear to be two main peaks in user activity: one in the morning/late morning and another, more prominent, in the late afternoon/early evening.
- **Lowest Activity in Early Morning**: User counts are consistently lowest across all seasons in the very early morning hours (approximately 1 AM to 5 AM).

Season-Specific Patterns:

- ### **Summer (Green Line)**:
- Generally shows the highest average user counts throughout the day, especially during peak hours.
- Morning peak is around 8 AM (over 300 users).
- Evening peak is the highest of all seasons, reaching over 400 users around 5 PM - 6 PM (17:00-18:00).
- Maintains high activity levels for a longer duration during the day.
- ### **Spring (Orange Line)**:
- Follows a very similar pattern to Summer, often just slightly below Summer's counts.
- Morning peak around 8 AM (around 300 users).

- Evening peak around 5 PM - 6 PM (17:00-18:00), also very high (close to 400 users).
- Fall (Blue Line):
- Shows a similar daily pattern to Spring and Summer but with slightly lower average counts overall, particularly during the evening peak.
- Morning peak around 8 AM (around 290 users).
- Evening peak around 5 PM - 6 PM (17:00-18:00), reaching around 380-390 users.
- Winter (Red Line):
- Consistently shows the lowest average user counts throughout the day compared to other seasons.
- Morning peak is lower, around 8 AM (around 250 users).
- Evening peak is also significantly lower, around 5 PM - 6 PM (17:00-18:00), reaching around 260-270 users.
- The overall curve is flatter, indicating less extreme fluctuations in user activity compared to warmer seasons.

Hours and days groub by mean of count of users

In [ ]:

```
hourAggregated = DataFrame(df.groupby(["hour","day"],sort=True)["count_of_users"].mean()).reset_index()
hourAggregated
```

Out[ ]:

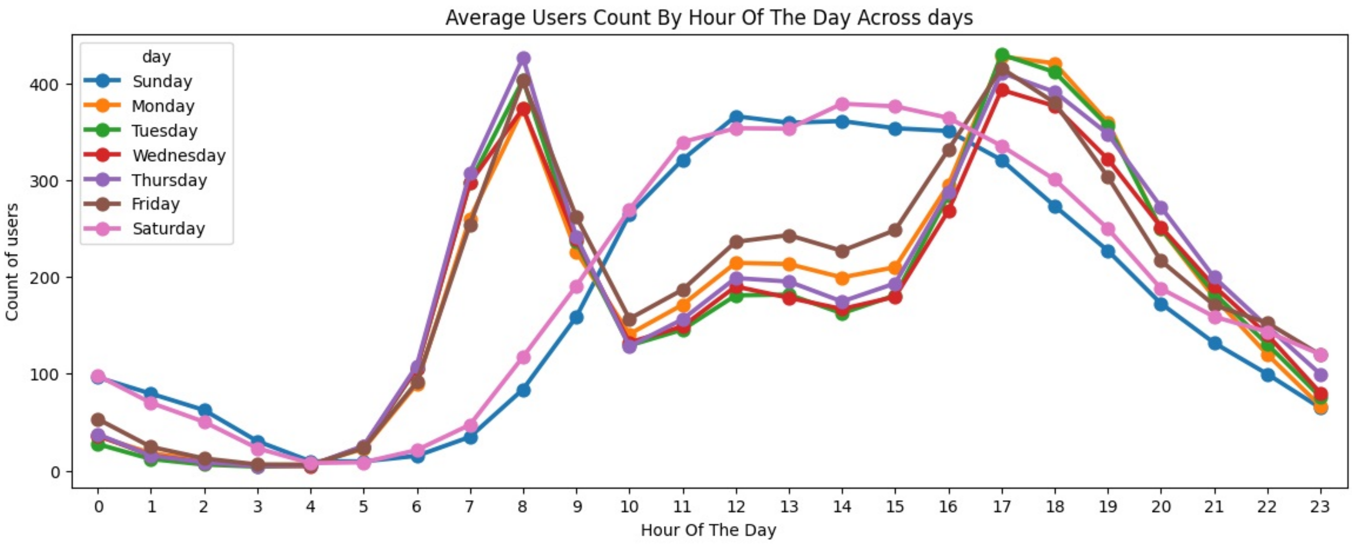
	hour	day	count_of_users
0	0	Friday	53.234375
1	0	Monday	35.492308
2	0	Saturday	98.212121
3	0	Sunday	96.227273
4	0	Thursday	37.476923
...	...	...	...
163	23	Saturday	120.030303
164	23	Sunday	64.757576
165	23	Thursday	99.630769
166	23	Tuesday	76.061538
167	23	Wednesday	80.138462

168 rows × 3 columns

In [ ]:

```
fig,ax= subplots()
fig.set_size_inches(14,5)

pointplot(x=hourAggregated["hour"], y=hourAggregated["count_of_users"],hue=hourAggregated["day"],hue_order=hue0
ax.set(xlabel='Hour Of The Day', ylabel='Count of users',title="Average Users Count By Hour Of The Day Across da
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\2714275970.py:4: UserWarning:

The `join` parameter is deprecated and will be removed in v0.15.0.

  pointplot(x=hourAggregated["hour"], y=hourAggregated["count_of_users"],hue=hourAggregated["day"],hue_order=hue
Order, data=hourAggregated, join=True,ax=ax)
```



- **Clear Daily Cycles**: All days exhibit a distinct daily cycle of user activity, with lows in the early morning and peaks during the day.
- **Weekend vs. Weekday Patterns**: There are noticeable differences between weekend (Saturday, Sunday) and weekday (Monday-Friday) patterns.
- **Early Morning Lows**: User counts are consistently very low across all days in the early morning hours (approximately 1 AM to 5 AM).

Day-Specific Patterns:

- **Weekdays (Monday-Friday)**:
- **Morning Peak**: Most weekdays show a prominent morning peak around 8 AM - 9 AM. This is likely associated with the start of the workday/school day. Thursday and Friday's morning peaks are slightly lower than Monday-Wednesday.
- **Mid-day Dip**: After the morning peak, there's a noticeable dip in user counts during late morning/early afternoon (around 9 AM - 3 PM).
- **Evening Peak**: All weekdays show a strong evening peak, typically around 5 PM - 7 PM (17:00-19:00). This peak is generally higher than the morning peak. Tuesday and Wednesday often show the highest evening peaks among weekdays, reaching over 400 users.
- **Decline**: User counts gradually decline from the evening peak into the late night.

Weekends (Saturday & Sunday):

- **Delayed Morning Activity**: Activity starts later in the morning compared to weekdays. The sharp rise in users begins around 7 AM - 8 AM, rather than 6 AM.
- **Single, Broader Peak**: Instead of distinct morning and evening peaks, weekends tend to have a single, broader peak of activity that spans from late morning through the afternoon.
- **Saturday**: Shows a strong, sustained high level of activity from around 10 AM to 6 PM (18:00), with a peak around 1 PM - 2 PM (13:00-14:00). The overall activity level is often lower than weekday evening peaks but higher than weekday mid-day dips.
- **Sunday**: Similar to Saturday, with a broad peak, but often with slightly lower overall counts than Saturday, especially in the afternoon.

Key Differences Summarized:

- **Weekday**: Characterized by two distinct peaks (morning and evening), with a mid-day dip. The evening peak is usually the highest.

- **Weekend**: Characterized by a later start to activity and a single, broader peak spanning the afternoon.

Hours groub by mean of subscribed users + non-subscribed users

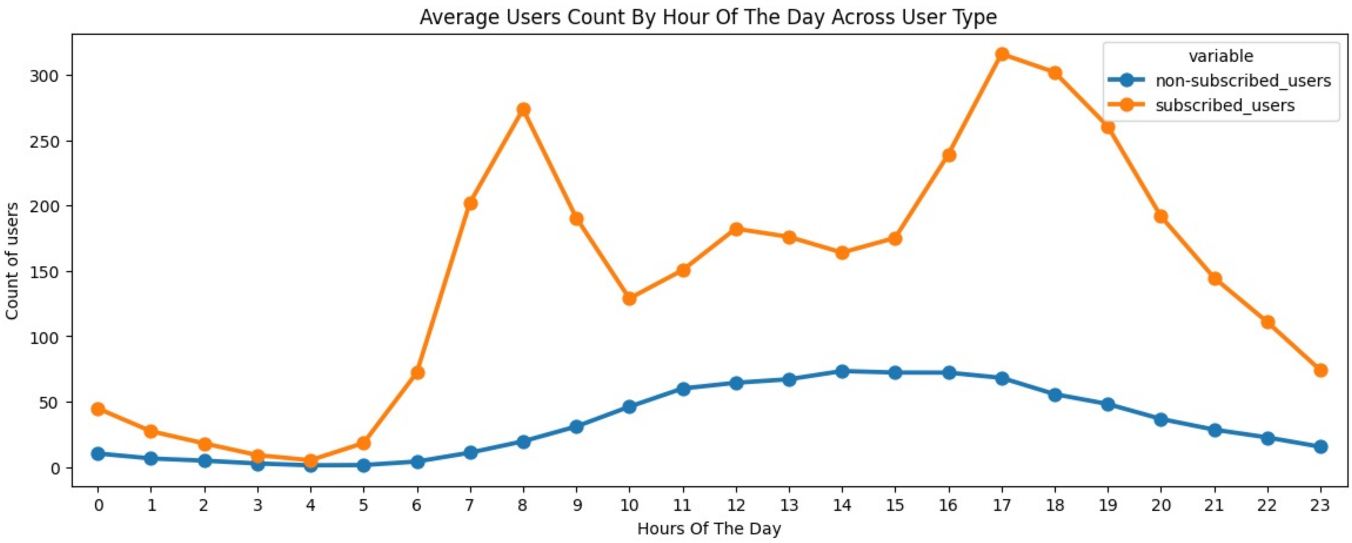
In [ ]:
```
hourTransformed = melt(df[["hour","non-subscribed_users","subscribed_users"]], id_vars=['hour'], value_vars=['n
hourAggregated = DataFrame(hourTransformed.groupby(["hour","variable"],sort=True)["value"].mean()).reset_index(
hourAggregated.head()
```

Out[ ]:

	hour	variable	value
0	0	non-subscribed_users	10.312088
1	0	subscribed_users	44.826374
2	1	non-subscribed_users	6.513216
3	1	subscribed_users	27.345815
4	2	non-subscribed_users	4.819196

In [ ]:
```
fig,ax= subplots()
fig.set_size_inches(14,5)

pointplot(x=hourAggregated["hour"], y=hourAggregated["value"],hue=hourAggregated["variable"],hue_order=["non-su
ax.set(xlabel='Hours Of The Day', ylabel='Count of users',title="Average Users Count By Hour Of The Day Across 
show()
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\1769763816.py:4: UserWarning:

The `join` parameter is deprecated and will be removed in v0.15.0.

  pointplot(x=hourAggregated["hour"], y=hourAggregated["value"],hue=hourAggregated["variable"],hue_order=["non-s
ubscribed_users","subscribed_users"], data=hourAggregated, join=True,ax=ax)
```



Subscribed Users Dominate Total Count:

- The 'subscribed_users' (orange line) consistently show a significantly higher average count at almost all hours compared to 'non-subscribed_users' (blue line). This indicates that the majority of the overall user activity is driven by subscribed users.
- **Distinct Daily Patterns for Each User Type:**
- **Non-subscribed Users (Blue Line)**:
- Low Activity in Early Morning: Very low counts from 0 to 5 AM.
- Gradual Increase: A slow, steady increase in average count from around 6 AM, peaking in the afternoon/early evening (around 3 PM - 5 PM / 15:00-17:00), reaching an average of about 70-75 users.
- Slow Decline: A gradual decline in counts through the late evening.
- The curve is relatively flat and does not show sharp peaks, suggesting a more consistent, less volatile activity pattern.
- Subscribed Users (Orange Line):
- Low Activity in Early Morning: Similar to non-subscribed users, very low counts from 0 to 5 AM.
- Sharp Morning Peak: A rapid increase in activity starting around 6 AM, leading to a prominent morning peak around 8 AM (over 270 users). This suggests a strong morning rush for subscribed users.
- Mid-day Dip: A noticeable dip in activity after the morning peak (around 9 AM - 10 AM).
- gher Evening Peak: Activity rises again, reaching an even higher evening peak around 5 PM - 6 PM (17:00-18:00), exceeding 300

users. This is the highest point of activity for subscribed users.

- Steep Decline: A relatively steep decline in counts through the late evening.

Comparison and Interpretation:

- Peak Times: Subscribed users exhibit two distinct peaks (morning and evening), which align with typical workday hours, suggesting they might be using the service for work-related or routine activities. Non-subscribed users have a single, broader, and much lower peak in the afternoon.
- Magnitude of Activity: Subscribed users contribute far more to the overall user count at any given hour.
- Behavioral Differences: The differing patterns suggest distinct behavioral characteristics between the two user types. Subscribed users show more structured, perhaps routine-driven, engagement, while non-subscribed users have a flatter, lower-level activity profile.

Transform data

In [ ]:
```
# Remove some columns
df = df.drop(['datetime', "date", 'time', 'non-subscribed_users', 'subscribed_users'], axis=1)
```

In [ ]:
```
df['year'] = df['year'].astype(int)
```

In [ ]:
```
object_data = df.select_dtypes('object')
non_object_data = df.select_dtypes('number')
```

In [ ]:
```
object_data
```

Out[ ]:

	season	weather	month	day	weekend	the_usual_time_periods_per_day	weather_feeling
0	Winter	Clear to Partly Cloudy	January	Saturday	Weekend	Midnight and After	Cool
1	Winter	Clear to Partly Cloudy	January	Saturday	Weekend	Midnight and After	Cool
2	Winter	Clear to Partly Cloudy	January	Saturday	Weekend	Midnight and After	Cool
3	Winter	Clear to Partly Cloudy	January	Saturday	Weekend	Midnight and After	Cool
4	Winter	Clear to Partly Cloudy	January	Saturday	Weekend	Midnight and After	Cool
...	...	...	...	...	...	...	...
10881	Fall	Clear to Partly Cloudy	December	Wednesday	No	Evening	Cool
10882	Fall	Clear to Partly Cloudy	December	Wednesday	No	Evening	Cool
10883	Fall	Clear to Partly Cloudy	December	Wednesday	No	Night	Cool
10884	Fall	Clear to Partly Cloudy	December	Wednesday	No	Night	Cool
10885	Fall	Clear to Partly Cloudy	December	Wednesday	No	Night	Cool

10586 rows × 7 columns

In [ ]:
```
object_data['weekend'].unique()
```

Out[ ]:
```
array(['Weekend', 'No'], dtype=object)
```

In [ ]:
```
object_data['weekend']= object_data['weekend'].replace('No',0)
object_data['weekend']= object_data['weekend'].replace('Weekend',1)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\470439055.py:2: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  object_data['weekend']= object_data['weekend'].replace('Weekend',1)
```

In [ ]:
```
object_data['weekend'].value_counts()
```

Out[ ]:
```
weekend
0    7470
1    3116
Name: count, dtype: int64
```

In [ ]:
```
object_data['weather'].unique()
```

Out[ ]:
```
array(['Clear to Partly Cloudy', 'Mist and Cloudy', 'Light Precipitation',
       'Severe Weather'], dtype=object)
```

In [ ]:
```
object_data['weather']= object_data['weather'].replace('Clear to Partly Cloudy',0)
object_data['weather']= object_data['weather'].replace('Mist and Cloudy',1)
object_data['weather']= object_data['weather'].replace('Light Precipitation',2)
object_data['weather']= object_data['weather'].replace('Severe Weather',3)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\1977241135.py:4: FutureWarning: Downcasting behavior in `replace
` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.inf
er_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', Tr
ue)`
  object_data['weather']= object_data['weather'].replace('Severe Weather',3)
```

In [ ]:
```
object_data['weather'].value_counts()
```

Out[ ]:
```
weather
0    6965
1    2770
2     850
3       1
Name: count, dtype: int64
```

In [ ]:
```
object_data['weather_feeling'].unique()
```

Out[ ]:
```
array(['Cool', 'Mild', 'Cold', 'Warm', 'Hot', 'Extremely Hot'],
      dtype=object)
```

In [ ]:
```
object_data['weather_feeling']= object_data['weather_feeling'].replace('Cold',0)
object_data['weather_feeling']= object_data['weather_feeling'].replace('Cool',1)
object_data['weather_feeling']= object_data['weather_feeling'].replace('Mild',2)
object_data['weather_feeling']= object_data['weather_feeling'].replace('Warm',3)
object_data['weather_feeling']= object_data['weather_feeling'].replace('Hot',4)
object_data['weather_feeling']= object_data['weather_feeling'].replace('Extremely Hot',5)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\4158022803.py:6: FutureWarning: Downcasting behavior in `replace
` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.inf
er_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', Tr
ue)`
  object_data['weather_feeling']= object_data['weather_feeling'].replace('Extremely Hot',5)
```

In [ ]:
```
object_data['weather_feeling'].value_counts()
```

Out[ ]:
```
weather_feeling
1    3307
2    2331
4    2127
3    1490
0     742
5     589
Name: count, dtype: int64
```

In [ ]:
```
object_data['month'].unique()
```

Out[ ]:
```
array(['January', 'February', 'March', 'April', 'May', 'June', 'July',
       'August', 'September', 'October', 'November', 'December'],
      dtype=object)
```

In [ ]:
```
object_data['month']= object_data['month'].replace('January',1)
object_data['month']= object_data['month'].replace('February',2)
object_data['month']= object_data['month'].replace('March',3)
object_data['month']= object_data['month'].replace('April',4)
object_data['month']= object_data['month'].replace('May',5)
object_data['month']= object_data['month'].replace('June',6)
object_data['month']= object_data['month'].replace('July',7)
object_data['month']= object_data['month'].replace('August',8)
object_data['month']= object_data['month'].replace('September',9)
object_data['month']= object_data['month'].replace('October',10)
object_data['month']= object_data['month'].replace('November',11)
object_data['month']= object_data['month'].replace('December',12)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\4200956100.py:12: FutureWarning: Downcasting behavior in `replac
e` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.in
fer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', T
rue)`
  object_data['month']= object_data['month'].replace('December',12)
```

In [ ]:
```
object_data['month'].value_counts()
```

Out[ ]:
```
month
2     901
12    899
11    895
3     885
7     885
1     884
5     883
4     881
8     872
10    871
6     870
9     860
Name: count, dtype: int64
```

In [ ]:
```
object_data['day'].unique()
```

Out[ ]:
```
array(['Saturday', 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
       'Friday'], dtype=object)
```

In [ ]:
```
object_data['day']= object_data['day'].replace('Monday',0)
object_data['day']= object_data['day'].replace('Tuesday',1)
object_data['day']= object_data['day'].replace('Wednesday',2)
object_data['day']= object_data['day'].replace('Thursday',3)
object_data['day']= object_data['day'].replace('Friday',4)
object_data['day']= object_data['day'].replace('Saturday',5)
object_data['day']= object_data['day'].replace('Sunday',6)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\3354629942.py:7: FutureWarning: Downcasting behavior in `replace
` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.inf
er_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', Tr
ue)`
  object_data['day']= object_data['day'].replace('Sunday',6)
```

In [ ]:
```
object_data['day'].value_counts()
```

Out[ ]:
```
day
6    1563
5    1553
0    1509
3    1494
2    1492
4    1491
1    1484
Name: count, dtype: int64
```

In [ ]:
```
object_data['season'].unique()
```

Out[ ]:
```
array(['Winter', 'Spring', 'Summer', 'Fall'], dtype=object)
```

In [ ]:
```
object_data['season']= object_data['season'].replace('Winter',0)
object_data['season']= object_data['season'].replace('Spring',1)
object_data['season']= object_data['season'].replace('Summer',2)
object_data['season']= object_data['season'].replace('Fall',3)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\570325207.py:4: FutureWarning: Downcasting behavior in `replace`
is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer
_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', True
)`
  object_data['season']= object_data['season'].replace('Fall',3)
```

In [ ]:
```
object_data['season'].value_counts()
```

Out[ ]:
```
season
0    2670
3    2665
1    2634
2    2617
Name: count, dtype: int64
```

In [ ]:
```
object_data['the_usual_time_periods_per_day'].unique()
```

Out[ ]:
```
array(['Midnight and After', 'Morning', 'Midday/Noon', 'Afternoon',
       'Evening', 'Night'], dtype=object)
```

In [ ]:
```
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Midnight a
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Morning',1
```

```
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Midday/No
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Afternoon
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Evening',
object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Night',5)
```

```
C:\Users\RPC\AppData\Local\Temp\ipykernel_17052\24245839.py:6: FutureWarning: Downcasting behavior in `replace`
is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer
_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', True
)`
  object_data['the_usual_time_periods_per_day']= object_data['the_usual_time_periods_per_day'].replace('Night',5
)
```

In [ ]:
```
object_data['the_usual_time_periods_per_day'].value_counts()
```

Out[ ]:
```
the_usual_time_periods_per_day
0    2684
1    2654
3    1711
5    1368
4    1282
2     887
Name: count, dtype: int64
```

In [ ]:
```
object_data
```

Out[ ]:

	season	weather	month	day	weekend	the_usual_time_periods_per_day	weather_feeling
0	0	0	1	5	1	0	1
1	0	0	1	5	1	0	1
2	0	0	1	5	1	0	1
3	0	0	1	5	1	0	1
4	0	0	1	5	1	0	1
...	...	...	...	...	...	...	...
10881	3	0	12	2	0	4	1
10882	3	0	12	2	0	4	1
10883	3	0	12	2	0	5	1
10884	3	0	12	2	0	5	1
10885	3	0	12	2	0	5	1

10586 rows × 7 columns

In [ ]:
```
object_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10586 entries, 0 to 10885
Data columns (total 7 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   season                          10586 non-null  int64
 1   weather                         10586 non-null  int64
 2   month                           10586 non-null  int64
 3   day                             10586 non-null  int64
 4   weekend                         10586 non-null  int64
 5   the_usual_time_periods_per_day  10586 non-null  int64
 6   weather_feeling                 10586 non-null  int64
dtypes: int64(7)
memory usage: 661.6 KB
```

In [ ]:
```
df = concat([non_object_data, object_data], axis=1)
df
```

Out[ ]:

	holiday	workingday	Actual_Temperature	Feels_Like_Temperature	humidity	windspeed	count_of_users	year	hour	seas
0	0	0	9.84	14.395	81	0.0000	16	2011	0	
1	0	0	9.02	13.635	80	0.0000	40	2011	1	
2	0	0	9.02	13.635	80	0.0000	32	2011	2	
3	0	0	9.84	14.395	75	0.0000	13	2011	3	
4	0	0	9.84	14.395	75	0.0000	1	2011	4	
...	...	...	...	...	...	...	...	...	...	
10881	0	1	15.58	19.695	50	26.0027	336	2012	19	
10882	0	1	14.76	17.425	57	15.0013	241	2012	20	
10883	0	1	13.94	15.910	61	15.0013	168	2012	21	
10884	0	1	13.94	17.425	61	6.0032	129	2012	22	
10885	0	1	13.12	16.665	66	8.9981	88	2012	23	

10586 rows × 16 columns

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10586 entries, 0 to 10885
Data columns (total 16 columns):
 #   Column                        Non-Null Count  Dtype  
---  ------                        --------------  -----  
 0   holiday                       10586 non-null  int64  
 1   workingday                    10586 non-null  int64  
 2   Actual_Temperature            10586 non-null  float64
 3   Feels_Like_Temperature        10586 non-null  float64
 4   humidity                      10586 non-null  int64  
 5   windspeed                     10586 non-null  float64
 6   count_of_users                10586 non-null  int64  
 7   year                          10586 non-null  int32  
 8   hour                          10586 non-null  int32  
 9   season                        10586 non-null  int64  
 10  weather                       10586 non-null  int64  
 11  month                         10586 non-null  int64  
 12  day                           10586 non-null  int64  
 13  weekend                       10586 non-null  int64  
 14  the_usual_time_periods_per_day  10586 non-null  int64  
 15  weather_feeling               10586 non-null  int64  
dtypes: float64(3), int32(2), int64(11)
memory usage: 1.3 MB
```

Model

Spliting Data to train and test

In [ ]:

```
x = df.drop('count_of_users', axis=1)
y = df['count_of_users']

x_train, x_test, y_train, y_test = train_test_split(x,y, train_size=0.80, random_state=123)

print(f'x_train : {x_train.shape}')
print(f'x_test : {x_test.shape}')
print('----------------------------')
print(f'y_train : {y_train.shape}')
print(f'y_test : {y_test.shape}')
```

```
x_train : (8468, 15)
x_test : (2118, 15)
----------------------------
y_train : (8468,)
y_test : (2118,)
```

Creating Model

In [ ]:
```
r_2=[]
rmse=[]
mae=[]

def reg(model):
    model.fit(x_train,y_train)
    pred = model.predict(x_test)

    R2 = r2_score(y_test,pred)
    RMSE = sqrt(mean_squared_error(y_test,pred))
    MAE = mean_absolute_error(y_test,pred)

    r_2.append(R2)
    rmse.append(RMSE)
    mae.append(MAE)
```

In [ ]:
```
LinearRegression_model = LinearRegression()
XGBRegressor_model = XGBRegressor()
RandomForestRegressor_model = RandomForestRegressor()
GradientBoostingRegressor_model = GradientBoostingRegressor()
```

In [ ]:
```
Algorithms = ['LinearRegression','XGBRegressor','RandomForestRegressor','GradientBoostingRegressor']
```

In [ ]:
```
reg(LinearRegression_model)
reg(XGBRegressor_model)
reg(RandomForestRegressor_model)
reg(GradientBoostingRegressor_model)
```

In [ ]:
```
result = DataFrame({'Algorithms':Algorithms,'R2':r_2,'rmse':rmse,'mae':mae})
result
```

Out[ ]:

	Algorithms	R2	rmse	mae
0	LinearRegression	0.423930	114.747856	88.345058
1	XGBRegressor	0.945005	35.454235	23.519283
2	RandomForestRegressor	0.940048	37.017592	23.728564
3	GradientBoostingRegressor	0.839117	60.640409	42.937868

XGBRegressor (Clearly the Best):

- Achieves the highest (R-score : 0.945) and the lowest RMSE (35.45) and MAE (23.51).
- This model is by far the most powerful when the data is on its original scale, demonstrating its superior ability to handle the complexities, non-linear relationships, and heteroscedasticity in the raw data.

RandomForestRegressor (Very Strong, Close Second to XGBoost):

- Comes in a very close second with a high (R-score : 0.9389) and low RMSE and MAE.
- Excellent performance also, reinforcing that ensemble tree-based models are highly effective for this data in its original form.

GradientBoostingRegressor (Good Performance):

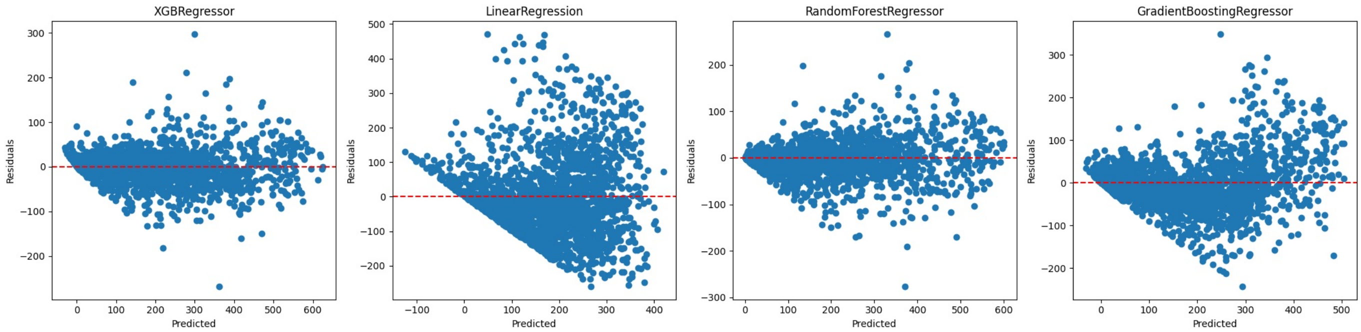
- Achieve good (R-score : 0.8699 and 0.839 respectively), but lag behind XGBoost and RandomForest.
- Strong models, but not as efficient as the two leading models in handling the original data.

LinearRegression (Significantly the Weakest):

- The worst by far (R-score : 0.423), with the highest RMSE (114.75) and MAE (88.34).
- This poor performance confirms that the relationship in the original "number of subscribers" data is not sufficiently linear, and there's a strong problem of heteroscedasticity, making the Linear Regression model unsuitable for this data in its raw form. This strongly justifies why the logarithmic transformation was so crucial for Linear Regression's later improved performance.

In [ ]:
```
fig, axes = subplots(1, 4, figsize=(20, 5))

residuals_xgb = y_test - XGBRegressor_model.predict(x_test)
axes[0].scatter(XGBRegressor_model.predict(x_test), residuals_xgb)
axes[0].axhline(y=0, color='red', linestyle='--')
axes[0].set_title("XGBRegressor")
axes[0].set_xlabel("Predicted")
```

```
axes[0].set_ylabel("Residuals")

residuals_lr = y_test - LinearRegression_model.predict(x_test)
axes[1].scatter(LinearRegression_model.predict(x_test), residuals_lr)
axes[1].axhline(y=0, color='red', linestyle='--')
axes[1].set_title("LinearRegression")
axes[1].set_xlabel("Predicted")
axes[1].set_ylabel("Residuals")

residuals_rf = y_test - RandomForestRegressor_model.predict(x_test)
axes[2].scatter(RandomForestRegressor_model.predict(x_test), residuals_rf)
axes[2].axhline(y=0, color='red', linestyle='--')
axes[2].set_title("RandomForestRegressor")
axes[2].set_xlabel("Predicted")
axes[2].set_ylabel("Residuals")

residuals_gb = y_test - GradientBoostingRegressor_model.predict(x_test)
axes[3].scatter(GradientBoostingRegressor_model.predict(x_test), residuals_gb)
axes[3].axhline(y=0, color='red', linestyle='--')
axes[3].set_title("GradientBoostingRegressor")
axes[3].set_xlabel("Predicted")
axes[3].set_ylabel("Residuals")

tight_layout()
show()
```



All presented regression models **(XGBRegressor, LinearRegression, RandomForestRegressor, GradientBoostingRegressor)** exhibit varying degrees of Heteroscedasticity (non-constant variance).

This phenomenon is particularly clear in the plots for **LinearRegression** and **GradientBoostingRegressor**, where the spread of prediction errors significantly increases as the predicted count of users grows.

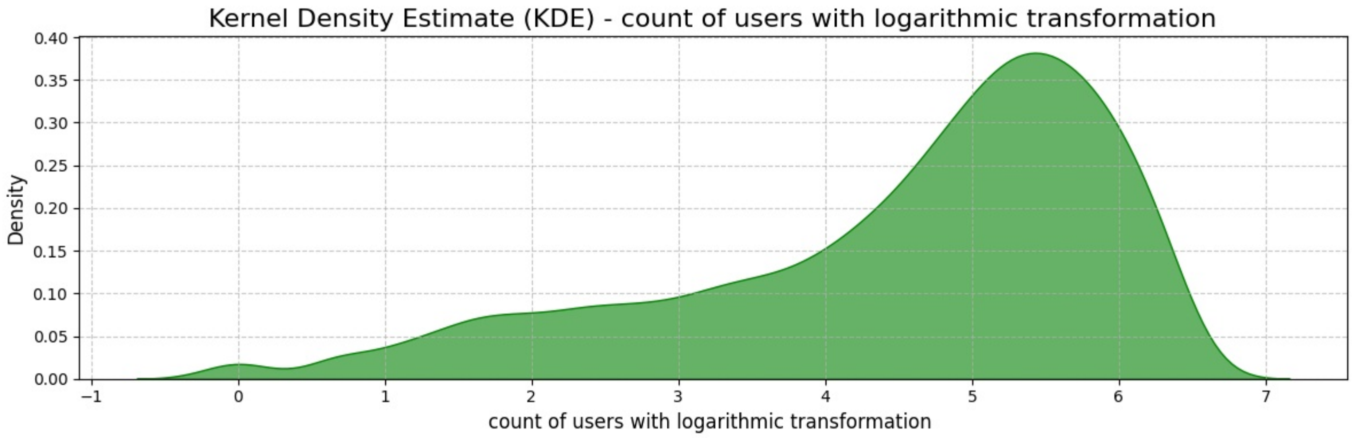
General Conclusion:

The fundamental issue common across all models is that the variance of the prediction errors increases with a higher predicted count of users. This means the model (regardless of its type) is less accurate and less reliable when predicting larger numbers of subscribers compared to smaller numbers.

To improve the performance of all models, and specifically to address this issue of heteroscedasticity, a logarithmic transformation to count of users

In [ ]:
```
y_transformed = log(y).replace(-inf,1e-6)
```

In [ ]:
```
figure(figsize=(12, 4))
kdeplot(y_transformed, fill=True, color='green', alpha=0.6)
title(f'Kernel Density Estimate (KDE) - count of users with logarithmic transformation', fontsize=16)
xlabel('count of users with logarithmic transformation ', fontsize=12)
ylabel('Density', fontsize=12)
grid(True, linestyle='--', alpha=0.7)
tight_layout()
show()
```



Shape of the Distribution:

- The distribution is now much closer to a normal (bell-shaped) distribution compared to what the original, untransformed "count of users" would likely have been (which would typically be heavily skewed to the right with a long tail).
- It has a clear peak (mode) somewhere between 5 and 6 on the transformed scale.
- There's still a slight left skew or a longer tail towards the lower values (left side of the peak), and a somewhat steeper drop-off on the right side. However, it's significantly less skewed than before.

Impact of Logarithmic Transformation:

- The transformation has successfully compressed the higher values of "count of users" and stretched out the lower values. This is precisely what's needed to mitigate heteroscedasticity.
- By making the distribution more symmetrical and less skewed, it's now more suitable for linear models and many other regression algorithms that often assume (or perform better with) normally distributed residuals and a more symmetrical target.

No Negative Values (Implied):

- While the X-axis starts at -1, the density curve effectively starts from just above 0 on the transformed scale (which corresponds to 0 or very small positive values on the original scale if np.log1p was used). This is good, as actual subscriber counts cannot be negative.

Spliting Data to train and test with y2

In [ ]:
```
x_train, x_test, y2_train, y2_test = train_test_split(x, y_transformed, train_size=0.80, random_state=123)
```

Creating Model by using y2 as new target

In [ ]:
```
r_2=[]
rmse=[]
mae=[]

def reg2(model):
    model.fit(x_train,y2_train)
    pred2 = model.predict(x_test)

    R2 = r2_score(y2_test,pred2)
    RMSE = sqrt(mean_squared_error(y2_test,pred2))
    MAE = mean_absolute_error(y2_test,pred2)

    r_2.append(R2)
    rmse.append(RMSE)
    mae.append(MAE)
```

In [ ]:
```
LinearRegression_model2 = LinearRegression()
XGBRegressor_model2 = XGBRegressor()
RandomForestRegressor_model2 = RandomForestRegressor()
GradientBoostingRegressor_model2 = GradientBoostingRegressor()
```

In [ ]:
```
Algorithms2 = ['LinearRegression','XGBRegressor','RandomForestRegressor','GradientBoostingRegressor']
```

In [ ]:
```
reg2(LinearRegression_model2)
reg2(XGBRegressor_model2)
reg2(RandomForestRegressor_model2)
```

```
reg2(GradientBoostingRegressor_model2)
```

In [ ]:
```
result2 = DataFrame({'Algorithms2':Algorithms2,'R2':r_2,'rmse':rmse,'mae':mae})
result2
```

Out[ ]:

	Algorithms2	R2	rmse	mae
0	LinearRegression	0.489662	1.008304	0.771287
1	XGBRegressor	0.949483	0.317236	0.209764
2	RandomForestRegressor	0.945295	0.330124	0.214289
3	GradientBoostingRegressor	0.912571	0.417341	0.302154

XGBRegressor (Best Performance):

- Achieves the highest R-score (0.94948) and the lowest RMSE (0.3172) and MAE (0.2097).
- This model is clearly the best performer after the logarithmic transformation. Its ability to explain nearly 95% of the variance in the log-transformed target variable, coupled with the lowest errors, makes it the most accurate choice for this type of data.

RandomForestRegressor (Very Strong, Second Place):

- Comes in a very close second to XGBRegressor (R-score : 0.94433) with very low RMSE (0.3330) and MAE (0.2160) values.
- This model also demonstrates exceptional strength and robustness in prediction, making it an excellent alternative to XGBRegressor.

DecisionTreeRegressor (Good Performance):

- Achieves a good R-score (0.91257), but with slightly higher RMSE (0.4173) and MAE (0.3021) values compared to the top two models.
- As a foundational tree-based model, it delivers strong and reliable performance after the transformation.

GradientBoostingRegressor (Good Performance):

- Its performance is good (R-score : 0.89264), with RMSE (0.4624) and MAE (0.2995) higher than DecisionTreeRegressor.
- An effective model, but it lags behind the other models in this specific context.

LinearRegression (Weakest Performance):

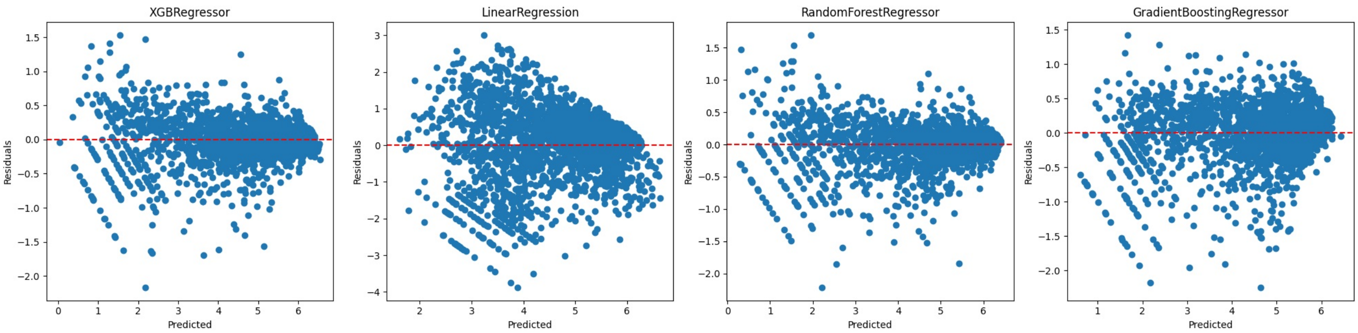
- Shows the weakest performance among all models (R-score : 0.48966), with the highest RMSE (1.0083) and MAE (0.7712).
- Despite the logarithmic transformation, this model remains the least effective in explaining variance and predicting the target variable in this case.

Overall Summary:

After the logarithmic transformation of the "number of subscribers" target variable, XGBRegressor and RandomForestRegressor clearly lead in performance. These models provide highly accurate and reliable predictions, making them the optimal choices for this data. Other models like DecisionTreeRegressor and GradientBoostingRegressor deliver good performance but are not at the same level as the top contenders. LinearRegression, in this context, is the least effective.

In [ ]:
```
fig, axes = subplots(1, 4, figsize=(20, 5))

residuals_xgb = y2_test - XGBRegressor_model2.predict(x_test)
axes[0].scatter(XGBRegressor_model2.predict(x_test), residuals_xgb)
axes[0].axhline(y=0, color='red', linestyle='--')
axes[0].set_title("XGBRegressor")
axes[0].set_xlabel("Predicted")
axes[0].set_ylabel("Residuals")

residuals_lr = y2_test - LinearRegression_model2.predict(x_test)
axes[1].scatter(LinearRegression_model2.predict(x_test), residuals_lr)
axes[1].axhline(y=0, color='red', linestyle='--')
axes[1].set_title("LinearRegression")
axes[1].set_xlabel("Predicted")
axes[1].set_ylabel("Residuals")

residuals_rf = y2_test - RandomForestRegressor_model2.predict(x_test)
axes[2].scatter(RandomForestRegressor_model2.predict(x_test), residuals_rf)
axes[2].axhline(y=0, color='red', linestyle='--')
axes[2].set_title("RandomForestRegressor")
axes[2].set_xlabel("Predicted")
axes[2].set_ylabel("Residuals")

residuals_gb = y2_test - GradientBoostingRegressor_model2.predict(x_test)
axes[3].scatter(GradientBoostingRegressor_model2.predict(x_test), residuals_gb)
axes[3].axhline(y=0, color='red', linestyle='--')
```

```python
axes[3].set_title("GradientBoostingRegressor")
axes[3].set_xlabel("Predicted")
axes[3].set_ylabel("Residuals")

tight_layout()
show()
```



XGBRegressor:

- **Scatter:** Points are well-scattered around the zero line (red dashed line).
- **Homoscedasticity:** Despite the transformation, there's still a noticeable fanning out (heteroscedasticity) of the residuals as predicted values increase. This means the variance of the error increases with higher predicted subscriber counts (on the logarithmic scale).
- **Bias:** There's a slight upward curve, indicating a minor tendency for underprediction at higher values.
- Generally excellent performance, but the heteroscedasticity issue has not been fully resolved.

LinearRegression:

- **Scatter**: Points are widely scattered and follow a clear curved pattern.
- **Homoscedasticity**: Shows severe heteroscedasticity. The fanning out is very clear, as the spread of residuals significantly increases with higher predicted values.
- **Bias:** There's a very strong curved pattern (non-linear relationship). Residuals are negative at lower predicted values (indicating overprediction) and become strongly positive at higher predicted values (indicating underprediction).
- This model is clearly unsuitable for the data, even after logarithmic transformation. The non-linear relationship remains a major problem.

RandomForestRegressor:

- **Scatter:** Similar to XGBRegressor, points are well-scattered around zero.
- **Homoscedasticity**: Shows signs of heteroscedasticity similar to XGBRegressor, with the spread increasing at higher predicted values.
- **Bias**: Residuals are generally centered around zero, with a slight upward trend at higher values.
- Performance is very close to XGBRegressor, but with persistent heteroscedasticity.

GradientBoostingRegressor:

- **Scatter**: Similar to RandomForest and XGBRegressor.
- **Homoscedasticity**: Exhibits heteroscedasticity with the spread widening at higher predicted values.
- **Bias**: There's a slight upward curve.
- Its performance is good, but not as ideal as XGBRegressor or RandomForestRegressor in achieving homoscedasticity.

General Summary of These Plots (Logarithmic Scale Residuals):

- **Partial Success of Log Transformation**: The logarithmic transformation has partially fulfilled its role: it succeeded in bringing the errors within a more manageable range (the Y-axis is now within a reasonable scale), which contributed to the improved R-score values we saw previously.
- **Persistent Heteroscedasticity**: However, all models, including the best performers (XGBRegressor and RandomForestRegressor), still exhibit noticeable heteroscedasticity (fanning out of residuals). This means that the model's accuracy varies depending on the magnitude of the prediction; it is less precise (or more variable) for larger predicted subscriber counts.
- **LinearRegression is Unsuitable**: The strong pattern in LinearRegression's residuals confirms that it is an unsuitable model for this data, even after logarithmic transformation.
- **Focus on Tree Models**: XGBRegressor and RandomForestRegressor remain the best choices as their residuals are closest to a random scatter, despite the persistent heteroscedasticity.

In [ ]:
```python
# !jupyter nbconvert --to html "Bake rental project.ipynb"
```