About YULU:-

Yulu is India's foremost provider of micro-mobility services, offering unique transportation solutions for daily commutes. Originating from a mission to alleviate traffic congestion in India, Yulu aims to provide safe commuting options through its user-friendly mobile app, facilitating shared, solo, and sustainable travel.

Yulu zones are strategically located at key points such as metro stations, bus stands, office complexes, residential areas, and corporate offices, ensuring seamless, affordable, and convenient first and last-mile connectivity.

Facing recent declines in revenue, Yulu has engaged a consulting firm to analyze the factors influencing the demand for their shared electric cycles. Specifically, they seek insights into the drivers of demand in the Indian market.

Problem Statement

The company wants to know:

- Which variables are significant in predicting the demand for shared electric cycles in the Indian market?
- How well those variables describe the electric cycle demands

Importing Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
import scipy.stats as stats
```

Loading the dataset

In [2]:
```
url = 'https://d2beiqkhq929f0.cloudfront.net/public_assets/assets/000/001/428/original
```

In [3]:
```
df = pd.read_csv(url)
```

Display the Shape of the dataset

Display statistical summary of numerical variables

In [4]:
```
print(df.shape)
print(df.describe())
```

```
(10886, 12)
             season       holiday    workingday       weather          temp  \
count  10886.000000  10886.000000  10886.000000  10886.000000  10886.00000
mean       2.506614      0.028569      0.680875      1.418427     20.23086
std        1.116174      0.166599      0.466159      0.633839      7.79159
min        1.000000      0.000000      0.000000      1.000000      0.82000
25%        2.000000      0.000000      0.000000      1.000000     13.94000
50%        3.000000      0.000000      1.000000      1.000000     20.50000
75%        4.000000      0.000000      1.000000      2.000000     26.24000
max        4.000000      1.000000      1.000000      4.000000     41.00000

              atemp      humidity     windspeed        casual    registered  \
count  10886.000000  10886.000000  10886.000000  10886.000000  10886.000000
mean      23.655084     61.886460     12.799395     36.021955    155.552177
std        8.474601     19.245033      8.164537     49.960477    151.039033
min        0.760000      0.000000      0.000000      0.000000      0.000000
25%       16.665000     47.000000      7.001500      4.000000     36.000000
50%       24.240000     62.000000     12.998000     17.000000    118.000000
75%       31.060000     77.000000     16.997900     49.000000    222.000000
max       45.455000    100.000000     56.996900    367.000000    886.000000

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

Columns in the Dataset

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

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

In [6]:
```
df.head()
```

Out[6]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	reg
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	

In [7]:
```
df.tail()
```

Out[7]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual
10881	2012-12-19 19:00:00	4	0	1	1	15.58	19.695	50	26.0027	7
10882	2012-12-19 20:00:00	4	0	1	1	14.76	17.425	57	15.0013	10
10883	2012-12-19 21:00:00	4	0	1	1	13.94	15.910	61	15.0013	4
10884	2012-12-19 22:00:00	4	0	1	1	13.94	17.425	61	6.0032	12
10885	2012-12-19 23:00:00	4	0	1	1	13.12	16.665	66	8.9981	4

Column Profiling

- datetime: datetime
- season: season (1: spring, 2: summer, 3: fall, 4: winter)
- holiday: whether day is a holiday or not (extracted from http://dchr.dc.gov/page/holiday-schedule)
- workingday: if day is neither weekend nor holiday is 1, otherwise is 0.

    - weather: 1: Clear, Few clouds, partly cloudy, partly cloudy 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp: temperature in Celsius
- atemp: feeling temperature in Celsius
- humidity: humidity
- windspeed: wind speed
- casual: count of casual users
- registered: count of registered users
- count: count of total rental bikes including both casual and registered

Check for Null Value and Duplicate Value

In [8]:
```
df.isna().sum()
```

Out[8]:
```
datetime      0
season        0
holiday       0
workingday    0
weather       0
temp          0
atemp         0
humidity      0
windspeed     0
casual        0
registered    0
count         0
dtype: int64
```

In [9]:
```
df.duplicated()
```

Out[9]:
```
0        False
1        False
2        False
3        False
4        False
         ...
10881    False
10882    False
10883    False
10884    False
10885    False
Length: 10886, dtype: bool
```

Datatype of the columns

In [10]:
```
df.dtypes
```

Out[10]:
```
datetime       object
season          int64
holiday         int64
workingday      int64
weather         int64
temp          float64
atemp         float64
humidity        int64
windspeed     float64
casual          int64
registered      int64
count           int64
dtype: object
```

Convert the datatype of the datetime column from object to datetime

In [11]:
```
df['datetime'] = pd.to_datetime(df['datetime'])
```

The time period for which the data is given can be determined by finding the minimum and maximum datetime values in the dataframe:

In [12]:
```
# Displaying the minimum value of the 'datetime' column
print("Minimum datetime value:", df['datetime'].min())

# Displaying the maximum value of the 'datetime' column
print("Maximum datetime value:", df['datetime'].max())

# Calculating the difference between the maximum and minimum datetime values
time_difference = df['datetime'].max() - df['datetime'].min()
print("Time difference:", time_difference)

# Adding a new column 'day' to store the day names from the 'datetime' column
df['day'] = df['datetime'].dt.day_name()
```

```
Minimum datetime value: 2011-01-01 00:00:00
Maximum datetime value: 2012-12-19 23:00:00
Time difference: 718 days 23:00:00
```

Set the 'datetime' column as the index of the df

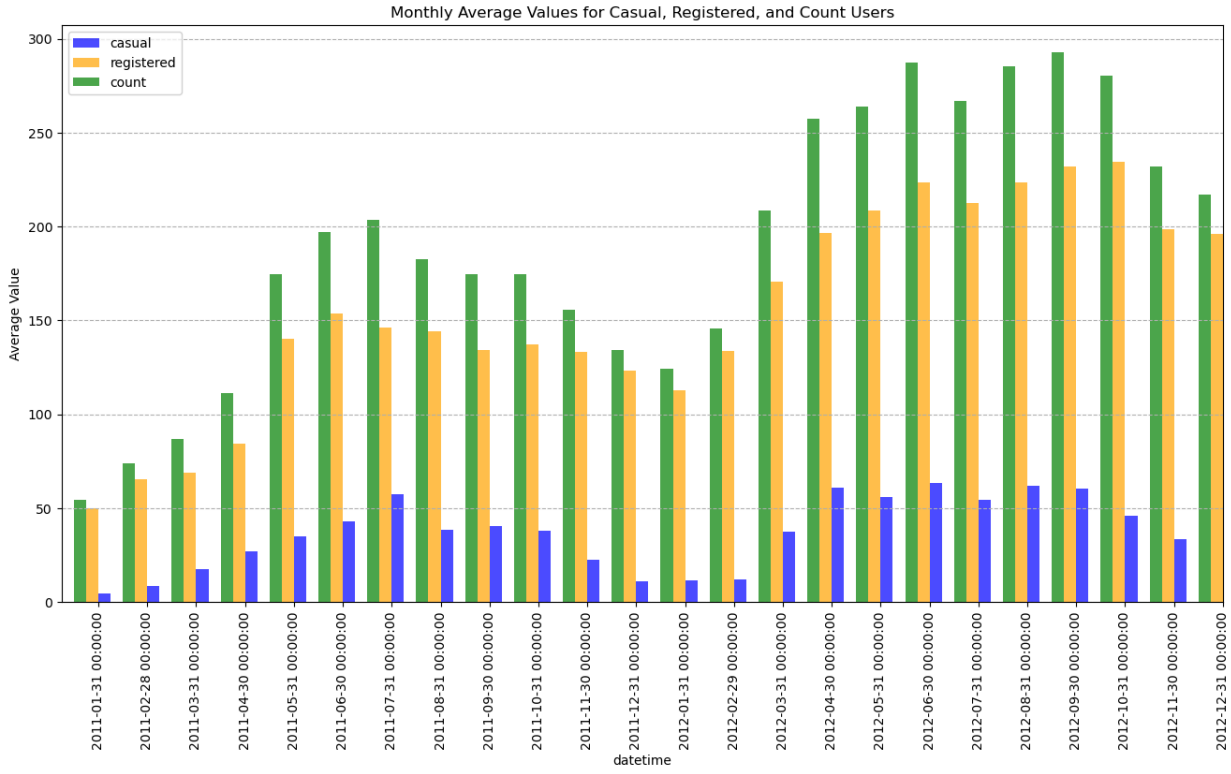
In [13]:
```
df.set_index('datetime', inplace=True)
```

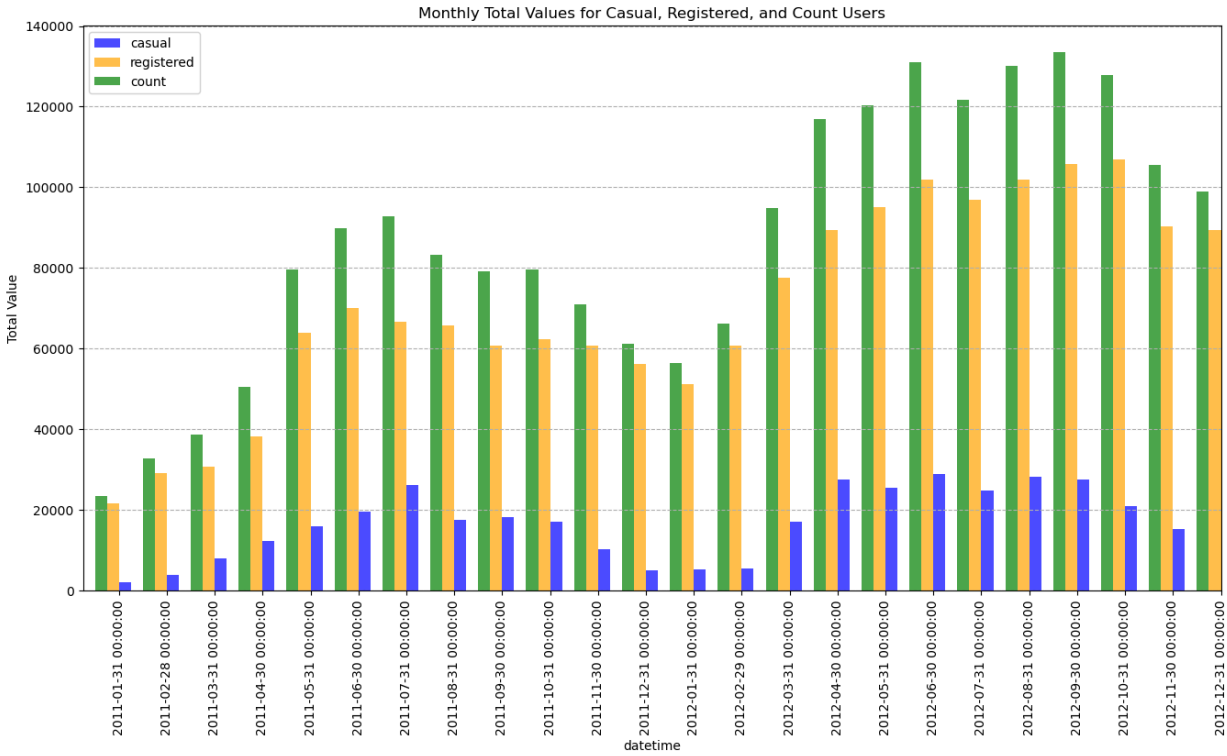
- By setting the 'datetime' column as the index, it allows for easier and more efficient access, filtering, and manipulation of the data based on the datetime values. It enables operations such as resampling, slicing by specific time periods, and applying time-based calculations.

Sliceing the data by time

In [14]:
```
# Visualizing monthly average values using a bar chart
plt.figure(figsize=(16, 8))

# Resampling the data on a monthly basis and calculating the mean value for 'casual',
df.resample('M')['casual'].mean().plot(kind='bar', color='blue', alpha=0.7, width=0.25
df.resample('M')['registered'].mean().plot(kind='bar', color='orange', alpha=0.7, widt
```

```
df.resample('M')['count'].mean().plot(kind='bar', color='green', alpha=0.7, width=0.25

plt.grid(axis='y', linestyle='--')  # Adding gridlines only along the y-axis
plt.ylabel('Average Value')
plt.title('Monthly Average Values for Casual, Registered, and Count Users')
plt.legend()
plt.show()  # Displaying the plot
# Visualizing monthly total values using a bar chart
plt.figure(figsize=(16, 8))

# Resampling the data on a monthly basis and calculating the sum of 'casual', 'registe
df.resample('M')['casual'].sum().plot(kind='bar', color='blue', alpha=0.7, width=0.25,
df.resample('M')['registered'].sum().plot(kind='bar', color='orange', alpha=0.7, width
df.resample('M')['count'].sum().plot(kind='bar', color='green', alpha=0.7, width=0.25,

plt.grid(axis='y', linestyle='--')  # Adding gridlines only along the y-axis
plt.ylabel('Total Value')
plt.title('Monthly Total Values for Casual, Registered, and Count Users')
plt.legend()
plt.show()  # Displaying the plot
```





Like to determine whether there's an increase in the average hourly count of rental bikes from 2011 to 2012.

In [15]:
```
# resampling the DataFrame by the year
df1 = df.resample('Y')['count'].mean().to_frame().reset_index()

# Create a new column 'prev_count' by shifting the 'count' column one position up
    # to compare the previous year's count with the current year's count
df1['prev_count'] = df1['count'].shift(1)

# Calculating the growth percentage of 'count' with respect to the 'count' of previous
df1['growth_percent'] = (df1['count'] - df1['prev_count']) * 100 / df1['prev_count']
df1
```

Out[15]:

	datetime	count	prev_count	growth_percent
0	2011-12-31	144.223349	NaN	NaN
1	2012-12-31	238.560944	144.223349	65.410764

- The data suggests a substantial growth in the count of rental bikes over the course of one year.
- The mean total hourly count of rental bikes is 144 for the year 2011.
- For the year 2012, the mean total hourly count increases to 239.
- This represents an annual growth rate of 65.41% in the demand for rental bikes on an hourly basis.
- The observed trend indicates positive growth and potentially signals a successful outcome or increasing demand for rental bikes.

In [16]:
```
df.reset_index(inplace = True)
```

What is the variation in the average hourly count of rental bikes for different months?

In [17]:
```
# Grouping the DataFrame by month
monthly_counts = df.groupby(by=df['datetime'].dt.month)['count'].mean().reset_index()
monthly_counts.rename(columns={'datetime': 'month'}, inplace=True)

# Create a new column 'prev_count' by shifting the 'count' column one position up
    # to compare the count of the previous month with the count of the current month
monthly_counts['prev_count'] = monthly_counts['count'].shift(1)

# Calculating the growth percentage of 'count' with respect to the count of the previc
monthly_counts['growth_percent'] = (monthly_counts['count'] - monthly_counts['prev_cou
monthly_counts.set_index('month', inplace=True)
monthly_counts
```

Out[17]:

month	count	prev_count	growth_percent
1	90.366516	NaN	NaN
2	110.003330	90.366516	21.730188
3	148.169811	110.003330	34.695751
4	184.160616	148.169811	24.290241
5	219.459430	184.160616	19.167406
6	242.031798	219.459430	10.285440
7	235.325658	242.031798	-2.770768
8	234.118421	235.325658	-0.513007
9	233.805281	234.118421	-0.133753
10	227.699232	233.805281	-2.611596
11	193.677278	227.699232	-14.941620
12	175.614035	193.677278	-9.326465

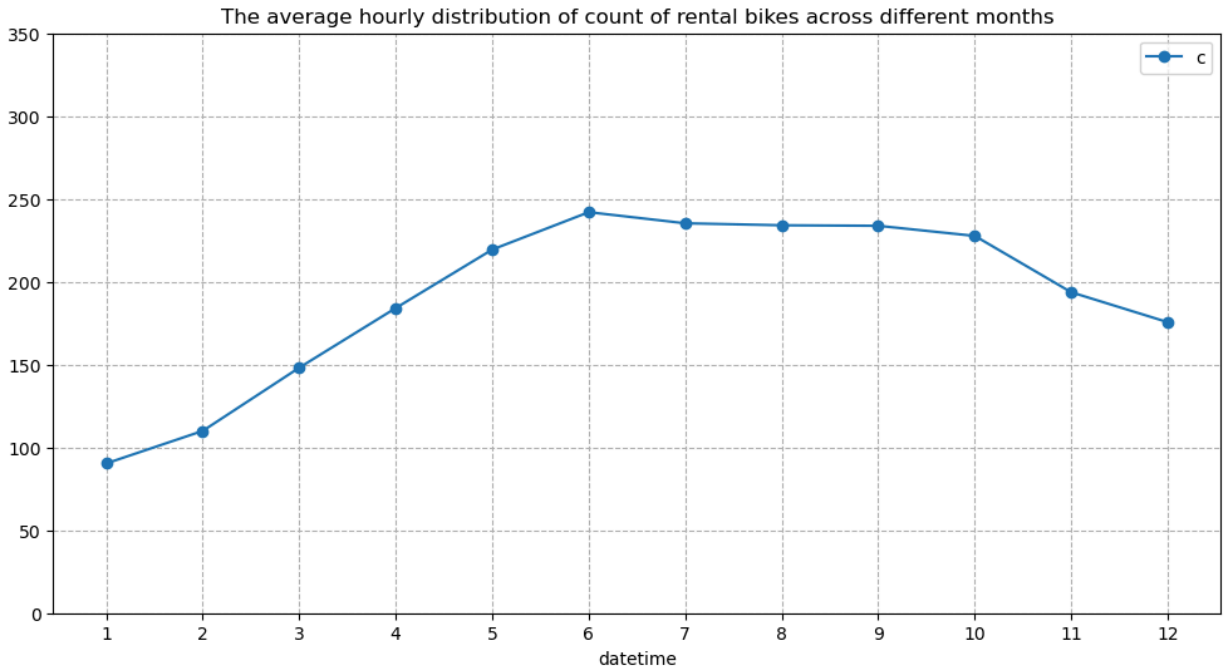
- The count of rental bikes exhibits an increasing trend from January to March, with a notable growth rate of 34.70% between February and March.
- The growth rate begins to stabilize from April to June, with relatively smaller growth rates observed during this period.
- From July to September, there is a slight decrease in the count of rental bikes, accompanied by negative growth rates.
- The count experiences a further decline from October to December, with the largest drop observed between October and November (-14.94%).

In [18]:
```
# Setting the figure size for the plot
plt.figure(figsize=(12, 6))
```

```
# Setting the title for the plot
plt.title("The average hourly distribution of count of rental bikes across different m

# Grouping the DataFrame by the month and calculating the mean of the 'count' column f
    # Ploting the line graph using markers ('o') to represent the average count per mo
df.groupby(by=df['datetime'].dt.month)['count'].mean().plot(kind='line', marker='o')

plt.ylim(0,)    # Setting the y-axis limits to start from zero
plt.xticks(np.arange(1, 13))   # Setting the x-ticks to represent the months from 1 to
plt.legend('count')    # Adding a legend to the plot for the 'count' line.
plt.yticks(np.arange(0, 400, 50))
# Adding gridlines to both the x and y axes with a dashed line style
plt.grid(axis='both', linestyle='--')
plt.plot()     # Displaing the plot.
```

Out[18]:

```
[]
```



- The average hourly count of rental bikes is highest in June, followed by July and August.
- Conversely, the average hourly count of rental bikes is lowest in January, followed by February and March.
- Overall, these trends suggest a seasonal pattern in the count of rental bikes. There's higher demand during the spring and summer months, a slight decline in the fall, and a further decrease in the winter months.
- Considering these patterns could be beneficial for the rental bike company in terms of resource allocation, marketing strategies, and operational planning throughout the year.

How is the average count of rental bikes distributed on an hourly basis within a single day?

In [19]:

```
# Grouping the DataFrame by the hour
hourly_counts = df.groupby(by=df['datetime'].dt.hour)['count'].mean().reset_index()
hourly_counts.rename(columns={'datetime': 'hour'}, inplace=True)

# Create a new column 'prev_count' by shifting the 'count' column one position up
    # to compare the count of the previous hour with the count of the current hour
```

```
hourly_counts['prev_count'] = hourly_counts['count'].shift(1)

# Calculating the growth percentage of 'count' with respect to the count of the previc
hourly_counts['growth_percent'] = (hourly_counts['count'] - hourly_counts['prev_count'
hourly_counts.set_index('hour', inplace=True)
hourly_counts
```

Out[19]:

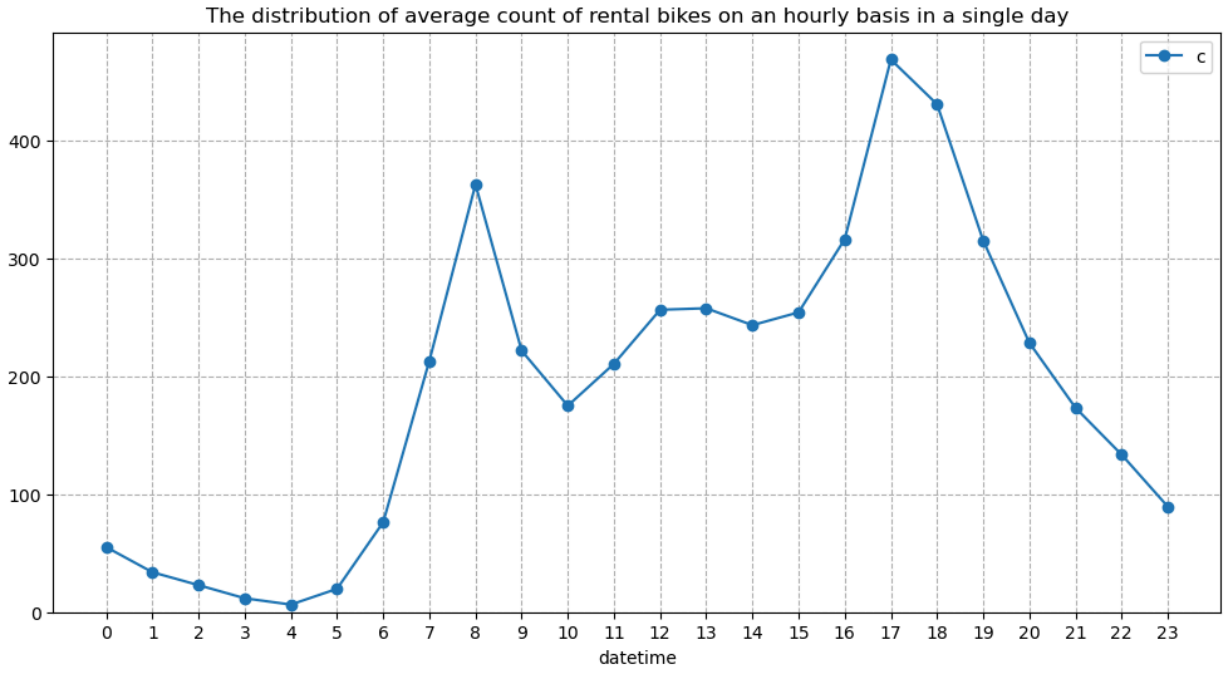
hour	count	prev_count	growth_percent
0	55.138462	NaN	NaN
1	33.859031	55.138462	-38.592718
2	22.899554	33.859031	-32.367959
3	11.757506	22.899554	-48.656179
4	6.407240	11.757506	-45.505110
5	19.767699	6.407240	208.521293
6	76.259341	19.767699	285.777526
7	213.116484	76.259341	179.462793
8	362.769231	213.116484	70.221104
9	221.780220	362.769231	-38.864655
10	175.092308	221.780220	-21.051432
11	210.674725	175.092308	20.322091
12	256.508772	210.674725	21.755835
13	257.787281	256.508772	0.498427
14	243.442982	257.787281	-5.564393
15	254.298246	243.442982	4.459058
16	316.372807	254.298246	24.410141
17	468.765351	316.372807	48.168661
18	430.859649	468.765351	-8.086285
19	315.278509	430.859649	-26.825705
20	228.517544	315.278509	-27.518833
21	173.370614	228.517544	-24.132471
22	133.576754	173.370614	-22.953059
23	89.508772	133.576754	-32.990757

- During the early morning hours (hours 0 to 5), there is a significant decrease in the count, with negative growth percentages ranging from -38.59% to -48.66%.
- However, starting from hour 5, there is a sudden increase in count, with a sharp positive growth percentage of 208.52% observed from hour 4 to hour 5.

- The count continues to rise significantly until reaching its peak at hour 17, with a growth percentage of 48.17% compared to the previous hour.
- After hour 17, there is a gradual decrease in count, with negative growth percentages ranging from -8.08% to -32.99% during the late evening and nighttime hours.

In [20]:
```
plt.figure(figsize = (12, 6))
plt.title("The distribution of average count of rental bikes on an hourly basis in a s
df.groupby(by = df['datetime'].dt.hour)['count'].mean().plot(kind = 'line', marker = '
plt.ylim(0,)
plt.xticks(np.arange(0, 24))
plt.legend('count')

plt.grid(axis = 'both', linestyle = '--')
plt.plot()
```

Out[20]:
```
[]
```



- The average count of rental bikes is highest at 5 PM, followed by 6 PM and 8 AM of the day.
- Conversely, the average count of rental bikes is lowest at 4 AM, followed by 3 AM and 5 AM of the day.
- These patterns indicate a distinct fluctuation in count throughout the day:

  1.Low counts during the early morning hours

  2.A sudden increase in the morning

  3.A peak count in the afternoon

  4.A gradual decline in the evening and nighttime.

Information about the Dataset

In [21]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype         
---  ------      --------------  -----         
 0   datetime    10886 non-null  datetime64[ns]
 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   temp        10886 non-null  float64       
 6   atemp       10886 non-null  float64       
 7   humidity    10886 non-null  int64         
 8   windspeed   10886 non-null  float64       
 9   casual      10886 non-null  int64         
 10  registered  10886 non-null  int64         
 11  count       10886 non-null  int64         
 12  day         10886 non-null  object        
dtypes: datetime64[ns](1), float64(3), int64(8), object(1)
memory usage: 1.1+ MB
```

- The dataframe currently requires approximately 1.1+ MB of memory usage.

Even though the current memory usage is relatively small, is there a method to further decrease it?

In [22]:
```
# Define a function to categorize seasons based on numerical values
def season_category(x):
    if x == 1:
        return 'spring'
    elif x == 2:
        return 'summer'
    elif x == 3:
        return 'fall'
    else:
        return 'winter'

# Apply the season_category function to the 'season' column in the DataFrame
df['season'] = df['season'].apply(season_category)
```

Optimizing Memory Usage of the DataFrame

In [23]:
```
# Updating dtype of season column
print('Memory usage of season column: ', df['season'].memory_usage())
df['season'] = df['season'].astype('category')
print('Updated Memory usage of season column: ', df['season'].memory_usage())

# Updating dtype of holiday column
print('Max value entry in holiday column: ', df['holiday'].max())
print('Memory usage of holiday column: ', df['holiday'].memory_usage())
df['holiday'] = df['holiday'].astype('category')
```

```
print('Updated Memory usage of holiday column: ', df['holiday'].memory_usage())

# Updating dtype of workingday column
print('Max value entry in workingday column: ', df['workingday'].max())
print('Memory usage of workingday column: ', df['workingday'].memory_usage())
df['workingday'] = df['workingday'].astype('category')
print('Updated Memory usage of workingday column: ', df['workingday'].memory_usage())

# Updating dtype of weather column
print('Max value entry in weather column: ', df['weather'].max())
print('Memory usage of weather column: ', df['weather'].memory_usage())
df['weather'] = df['weather'].astype('category')
print('Updated Memory usage of weather column: ', df['weather'].memory_usage())

# Updating dtype of temp column
print('Max value entry in temp column: ', df['temp'].max())
print('Memory usage of temp column: ', df['temp'].memory_usage())
df['temp'] = df['temp'].astype('float32')
print('Updated Memory usage of temp column: ', df['temp'].memory_usage())

# Updating dtype of atemp column
print('Max value entry in atemp column: ', df['atemp'].max())
print('Memory usage of atemp column: ', df['atemp'].memory_usage())
df['atemp'] = df['atemp'].astype('float32')
print('Updated Memory usage of atemp column: ', df['atemp'].memory_usage())

# Updating dtype of humidity column
print('Max value entry in humidity column: ', df['humidity'].max())
print('Memory usage of humidity column: ', df['humidity'].memory_usage())
df['humidity'] = df['humidity'].astype('int8')
print('Updated Memory usage of humidity column: ', df['humidity'].memory_usage())

# Updating dtype of windspeed column
print('Max value entry in windspeed column: ', df['windspeed'].max())
print('Memory usage of windspeed column: ', df['windspeed'].memory_usage())
df['windspeed'] = df['windspeed'].astype('float32')
print('Updated Memory usage of windspeed column: ', df['windspeed'].memory_usage())

# Updating dtype of casual column
print('Max value entry in casual column: ', df['casual'].max())
print('Memory usage of casual column: ', df['casual'].memory_usage())
df['casual'] = df['casual'].astype('int16')
print('Updated Memory usage of casual column: ', df['casual'].memory_usage())

# Updating dtype of registered column
print('Max value entry in registered column: ', df['registered'].max())
print('Memory usage of registered column: ', df['registered'].memory_usage())
df['registered'] = df['registered'].astype('int16')
print('Updated Memory usage of registered column: ', df['registered'].memory_usage())

# Updating dtype of count column
print('Max value entry in count column: ', df['count'].max())
print('Memory usage of count column: ', df['count'].memory_usage())
df['count'] = df['count'].astype('int16')
print('Updated Memory usage of count column: ', df['count'].memory_usage())
```

```
Memory usage of season column:  87220
Updated Memory usage of season column:  11222
Max value entry in holiday column:  1
Memory usage of holiday column:  87220
Updated Memory usage of holiday column:  11142
Max value entry in workingday column:  1
Memory usage of workingday column:  87220
Updated Memory usage of workingday column:  11142
Max value entry in weather column:  4
Memory usage of weather column:  87220
Updated Memory usage of weather column:  11222
Max value entry in temp column:  41.0
Memory usage of temp column:  87220
Updated Memory usage of temp column:  43676
Max value entry in atemp column:  45.455
Memory usage of atemp column:  87220
Updated Memory usage of atemp column:  43676
Max value entry in humidity column:  100
Memory usage of humidity column:  87220
Updated Memory usage of humidity column:  11018
Max value entry in windspeed column:  56.9969
Memory usage of windspeed column:  87220
Updated Memory usage of windspeed column:  43676
Max value entry in casual column:  367
Memory usage of casual column:  87220
Updated Memory usage of casual column:  21904
Max value entry in registered column:  886
Memory usage of registered column:  87220
Updated Memory usage of registered column:  21904
Max value entry in count column:  977
Memory usage of count column:  87220
Updated Memory usage of count column:  21904
```

In [24]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  datetime64[ns]
 1   season      10886 non-null  category
 2   holiday     10886 non-null  category
 3   workingday  10886 non-null  category
 4   weather     10886 non-null  category
 5   temp        10886 non-null  float32
 6   atemp       10886 non-null  float32
 7   humidity    10886 non-null  int8
 8   windspeed   10886 non-null  float32
 9   casual      10886 non-null  int16
 10  registered  10886 non-null  int16
 11  count       10886 non-null  int16
 12  day         10886 non-null  object
dtypes: category(4), datetime64[ns](1), float32(3), int16(3), int8(1), object(1)
memory usage: 415.4+ KB
```

- Previously, the dataset occupied 1.1+ MB of memory, but now it has been reduced to 415.2+ KB, resulting in approximately a 63.17% reduction in memory usage.

Description of the dataset

In [25]:
```
df.describe()
```

Out[25]:

	temp	atemp	humidity	windspeed	casual	registered	co
count	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000	10886.000
mean	20.230862	23.655085	61.886460	12.799396	36.021955	155.552177	191.574
std	7.791600	8.474654	19.245033	8.164592	49.960477	151.039033	181.144
min	0.820000	0.760000	0.000000	0.000000	0.000000	0.000000	1.000
25%	13.940000	16.665001	47.000000	7.001500	4.000000	36.000000	42.000
50%	20.500000	24.240000	62.000000	12.998000	17.000000	118.000000	145.000
75%	26.240000	31.059999	77.000000	16.997900	49.000000	222.000000	284.000
max	41.000000	45.455002	100.000000	56.996899	367.000000	886.000000	977.000

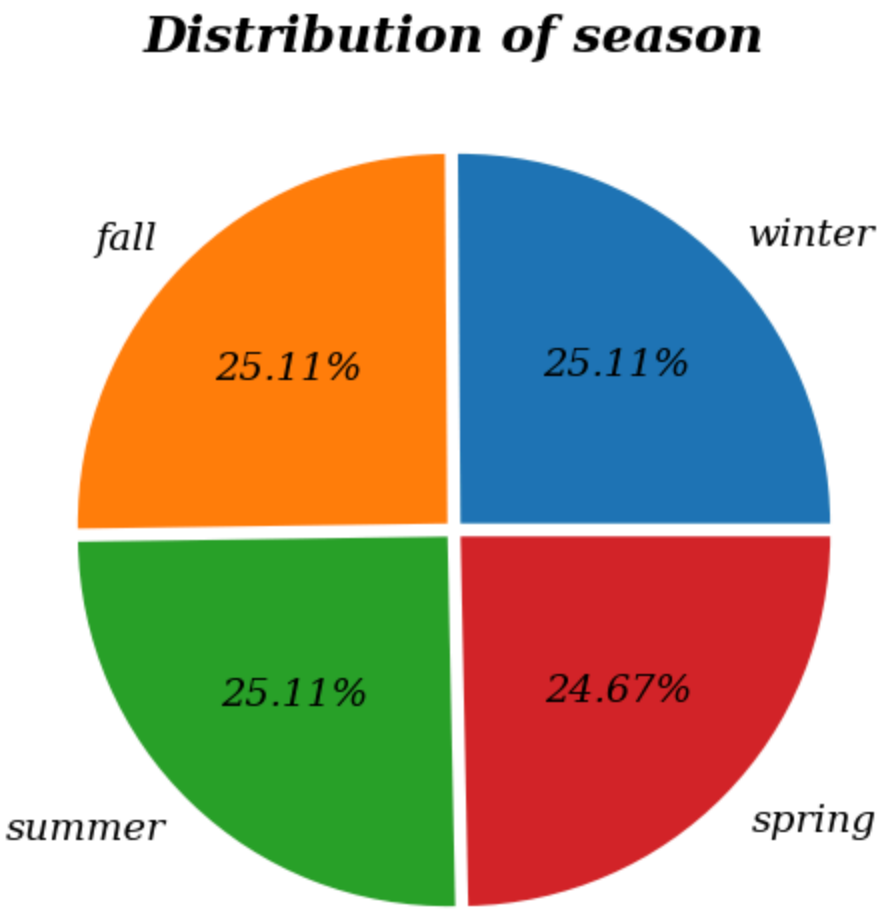
- These statistics offer insights into the average, variability, and extent of the numerical attributes within the dataset.

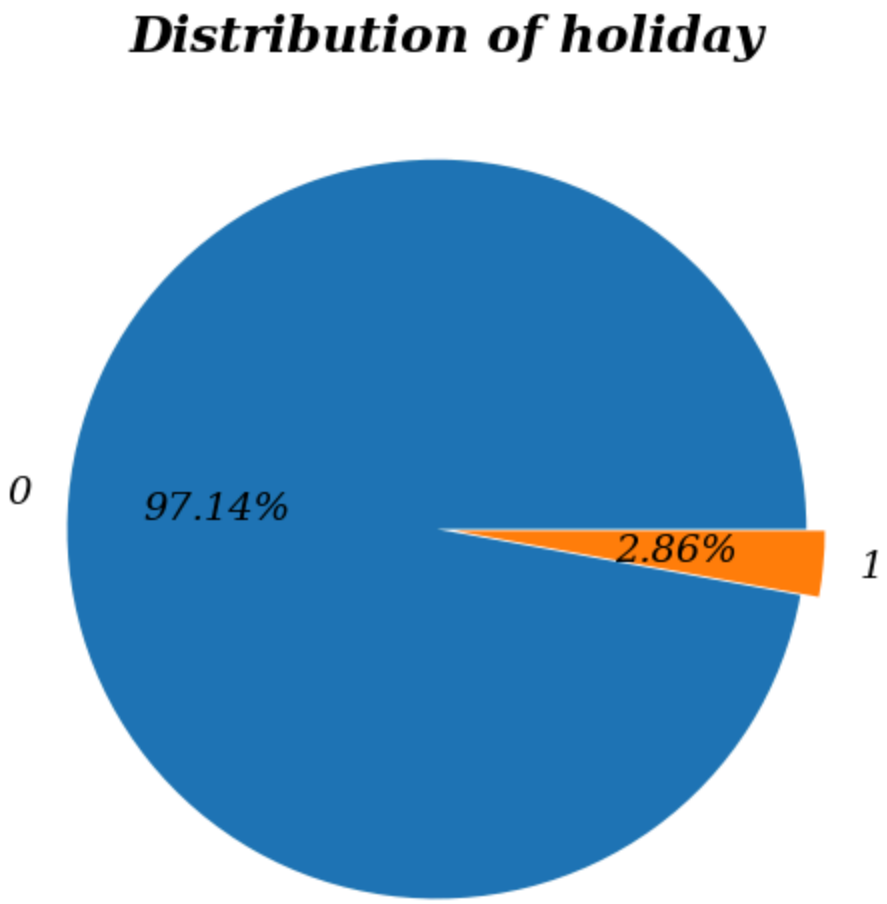
In [26]:
```
# Percentage distribution of 'season' column
season_distribution = np.round(df['season'].value_counts(normalize=True) * 100, 2)
print("Percentage distribution of 'season' column:")
print(season_distribution)

# Percentage distribution of 'holiday' column
holiday_distribution = np.round(df['holiday'].value_counts(normalize=True) * 100, 2)
print("\nPercentage distribution of 'holiday' column:")
print(holiday_distribution)

# Percentage distribution of 'workingday' column
workingday_distribution = np.round(df['workingday'].value_counts(normalize=True) * 100
print("\nPercentage distribution of 'workingday' column:")
print(workingday_distribution)

# Percentage distribution of 'weather' column
weather_distribution = np.round(df['weather'].value_counts(normalize=True) * 100, 2)
print("\nPercentage distribution of 'weather' column:")
print(weather_distribution)
```

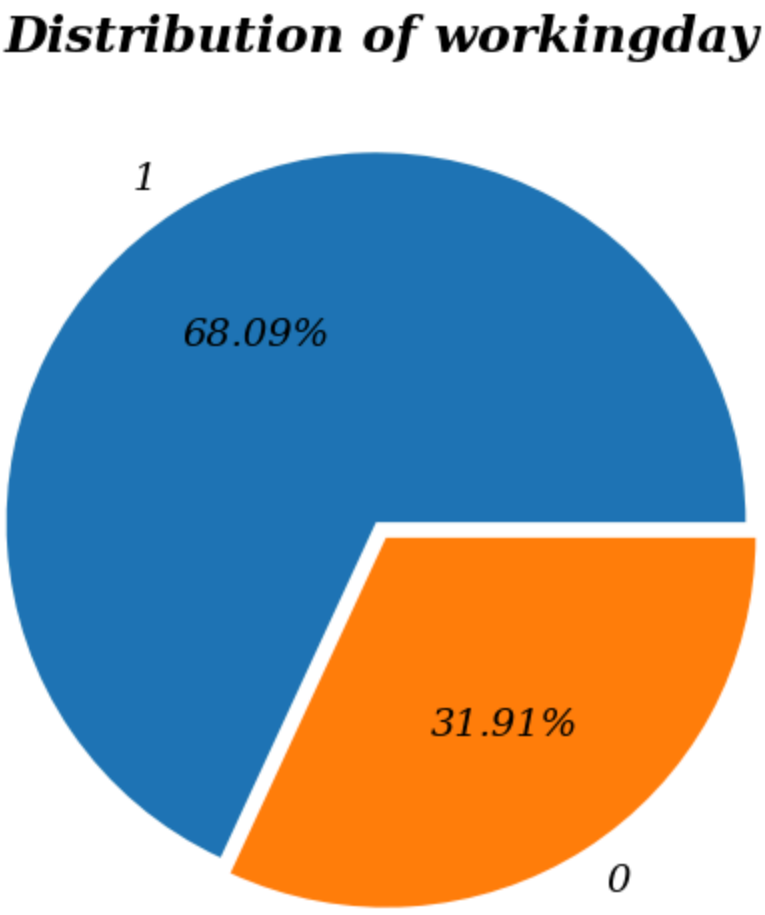
```
Percentage distribution of 'season' column:
winter    25.11
fall      25.11
summer    25.11
spring    24.67
Name: season, dtype: float64

Percentage distribution of 'holiday' column:
0    97.14
1     2.86
Name: holiday, dtype: float64

Percentage distribution of 'workingday' column:
1    68.09
0    31.91
Name: workingday, dtype: float64

Percentage distribution of 'weather' column:
1    66.07
2    26.03
3     7.89
4     0.01
Name: weather, dtype: float64
```

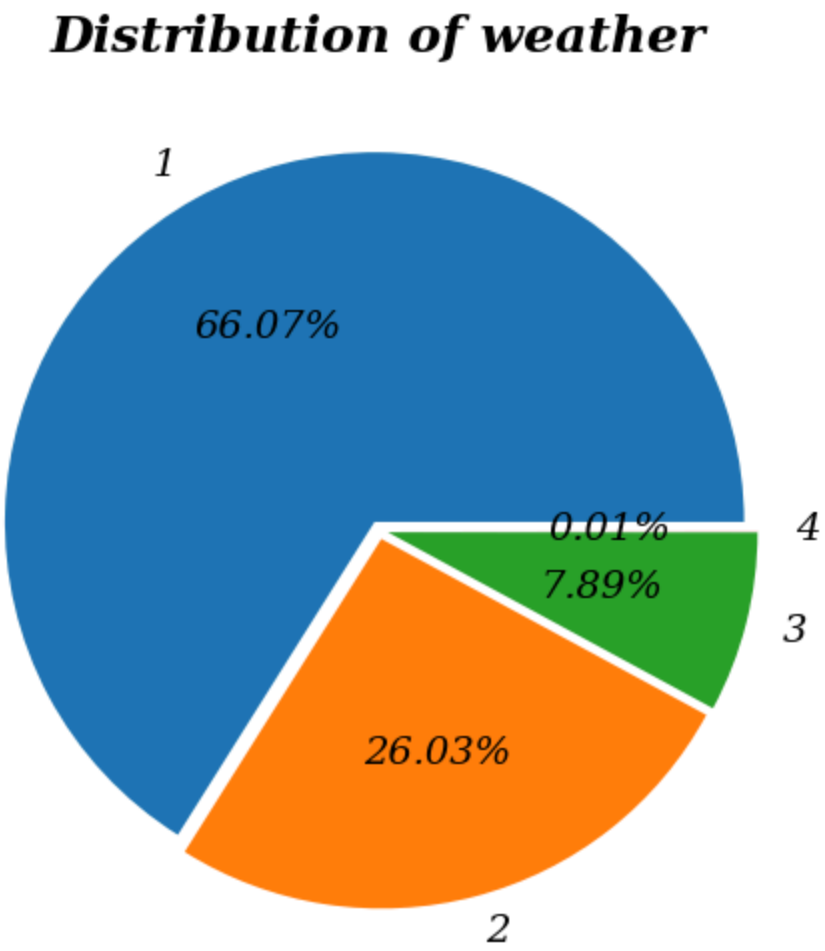
- Calculate the percentage distribution of each categorical column in the DataFrame.

In [27]:
```
# Function to create a pie chart
def create_pie_chart(data, title):
    plt.figure(figsize=(6, 6))
    plt.title(title, fontdict={'fontsize': 18, 'fontweight': 600, 'fontstyle': 'obliqu
    plt.pie(x=data.values, explode=[0.025] * len(data), labels=data.index, autopct='%.
    plt.plot()

# Distribution of seasons
season_distribution = np.round(df['season'].value_counts(normalize=True) * 100, 2)
create_pie_chart(season_distribution, 'Distribution of season')

# Distribution of holiday
holiday_distribution = np.round(df['holiday'].value_counts(normalize=True) * 100, 2)
create_pie_chart(holiday_distribution, 'Distribution of holiday')

# Distribution of workingday
workingday_distribution = np.round(df['workingday'].value_counts(normalize=True) * 100
create_pie_chart(workingday_distribution, 'Distribution of workingday')

# Distribution of weather
weather_distribution = np.round(df['weather'].value_counts(normalize=True) * 100, 2)
create_pie_chart(weather_distribution, 'Distribution of weather')

plt.show()
```


Distribution of season
fall
25.11%
winter
25.11%
summer
25.11%
spring
24.67%


Distribution of holiday
0
97.14%
2.86%
1

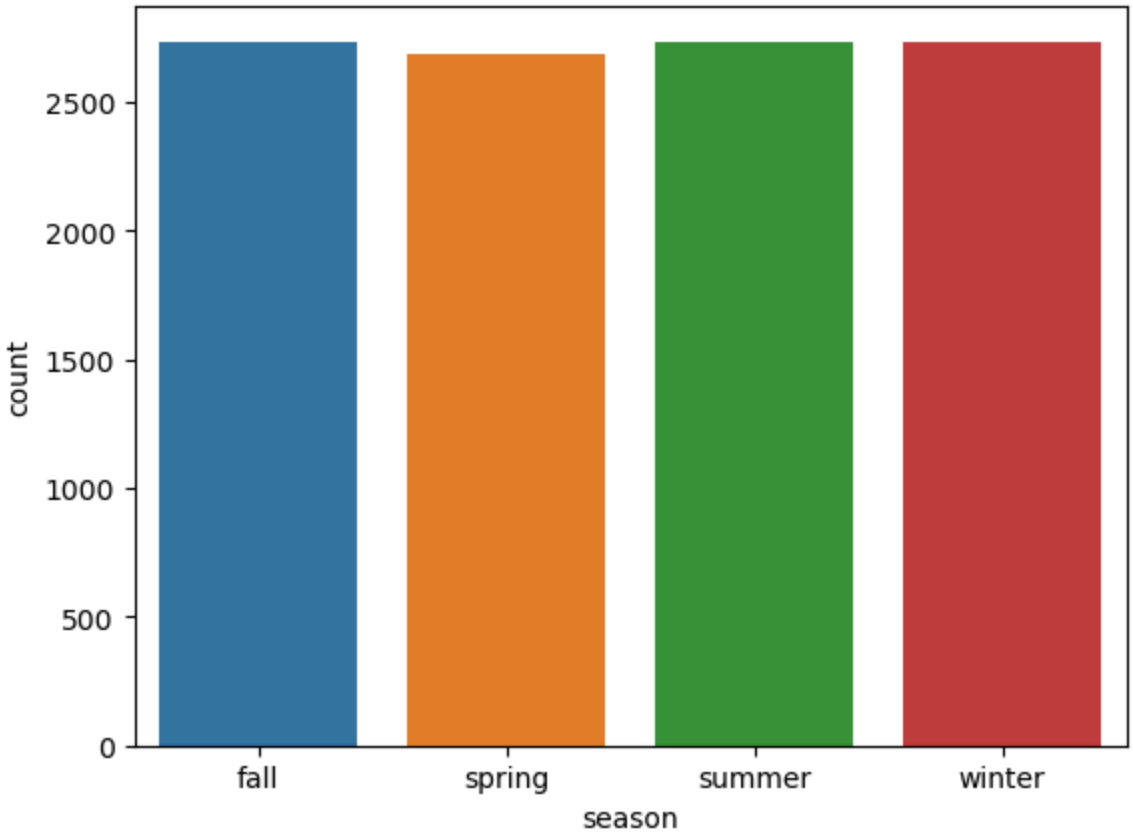

Distribution of workingday
1
68.09%
31.91%
0

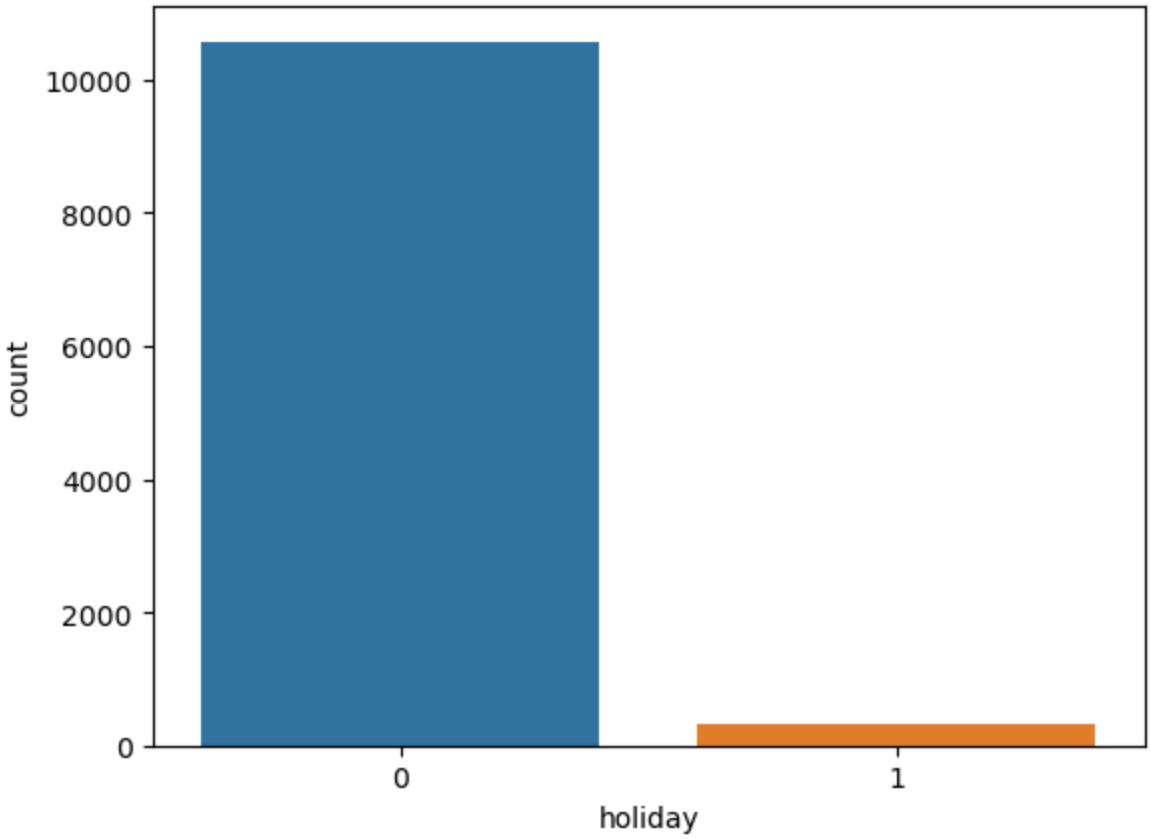


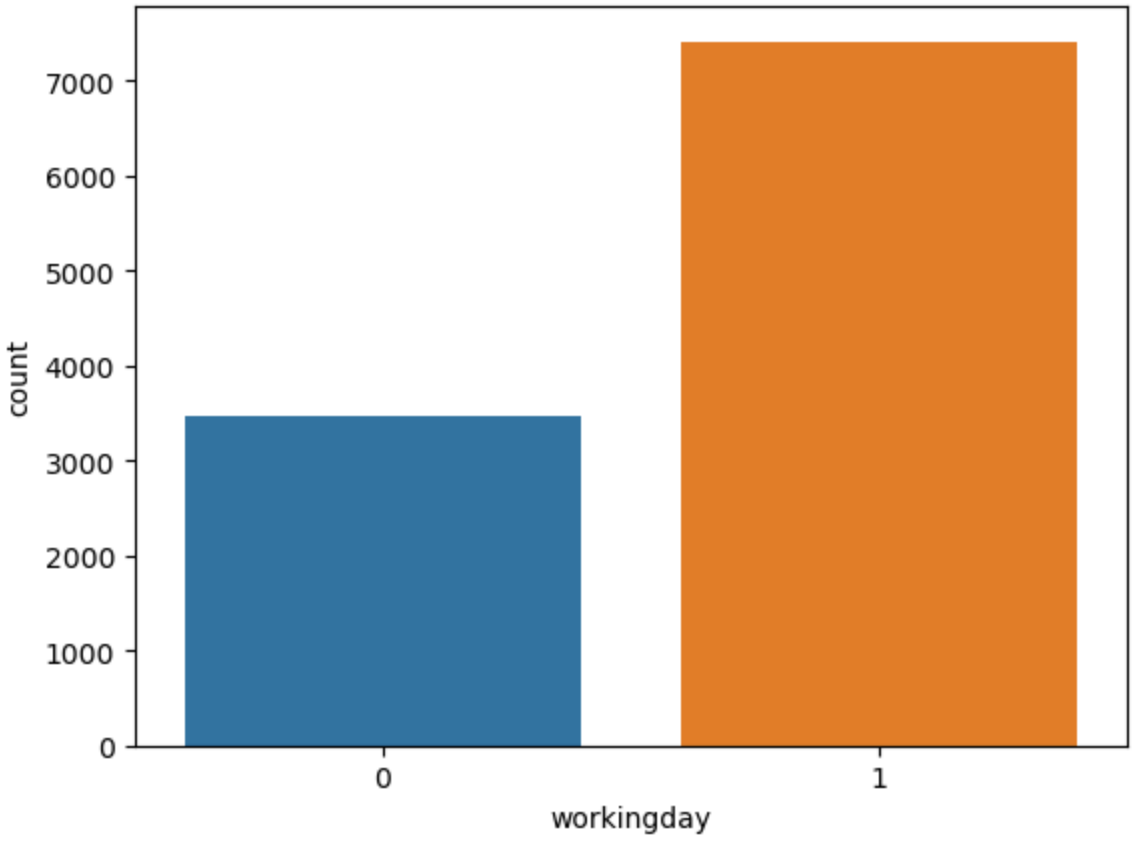
- Creates pie charts to show how seasons, holidays, working days, and weather are distributed in the dataset. Each chart has a clear title and style for easy understanding.

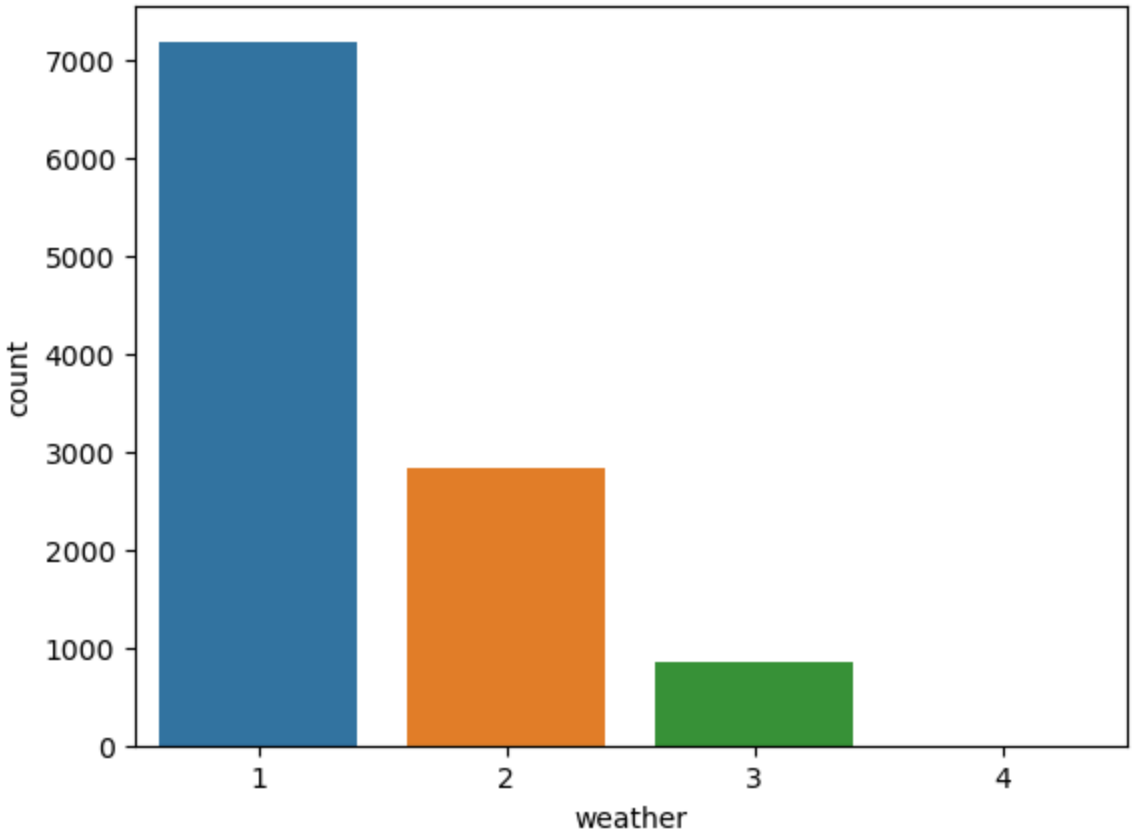
Univariate Analysis

In [28]:
```
# Distribution of seasons
sns.countplot(data=df, x='season')
plt.show()

# Distribution of holidays
sns.countplot(data=df, x='holiday')
plt.show()

# Distribution of working days
sns.countplot(data=df, x='workingday')
plt.show()

# Distribution of weather conditions
sns.countplot(data=df, x='weather')
plt.show()
```


2500
2000
1500
1000
500
0
count
fall
spring
summer
winter
season


10000
8000
6000
4000
2000
0
count
0
1
holiday



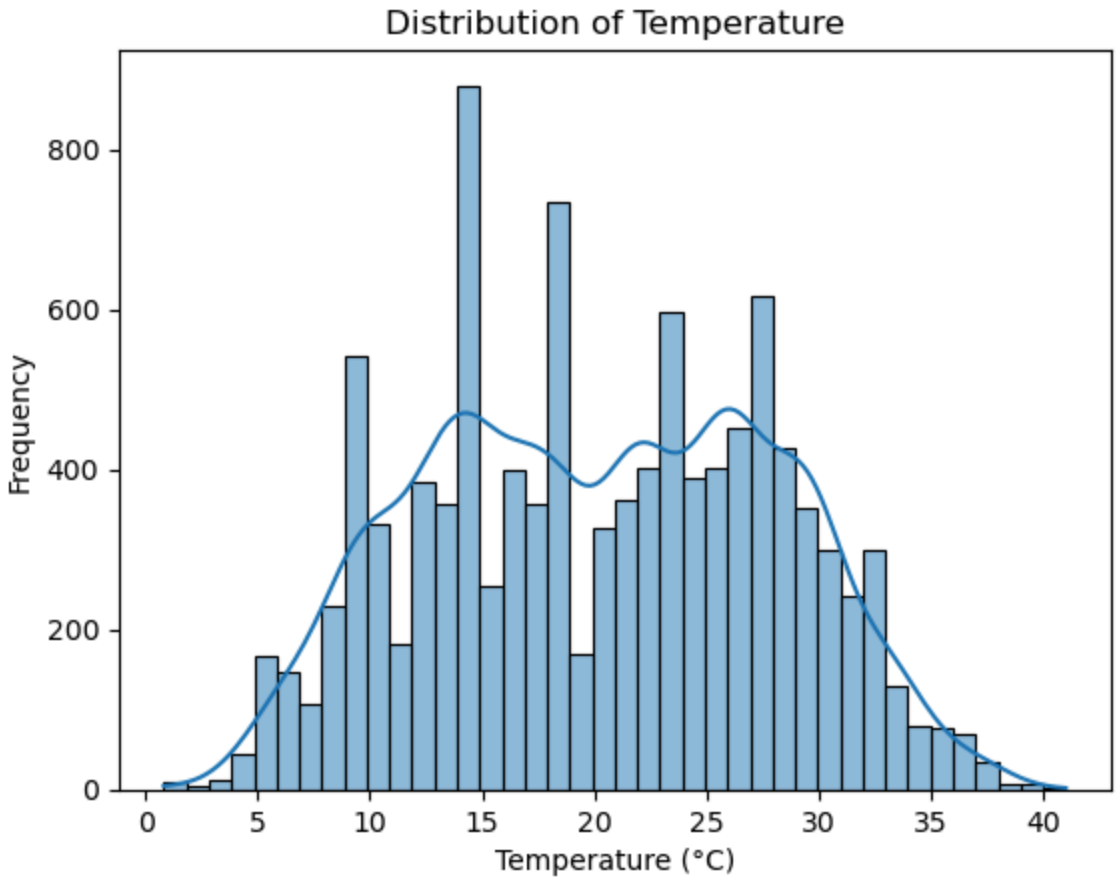


- The code utilizes the Seaborn library to create count plots for visualizing the distribution of categorical variables in the dataset.

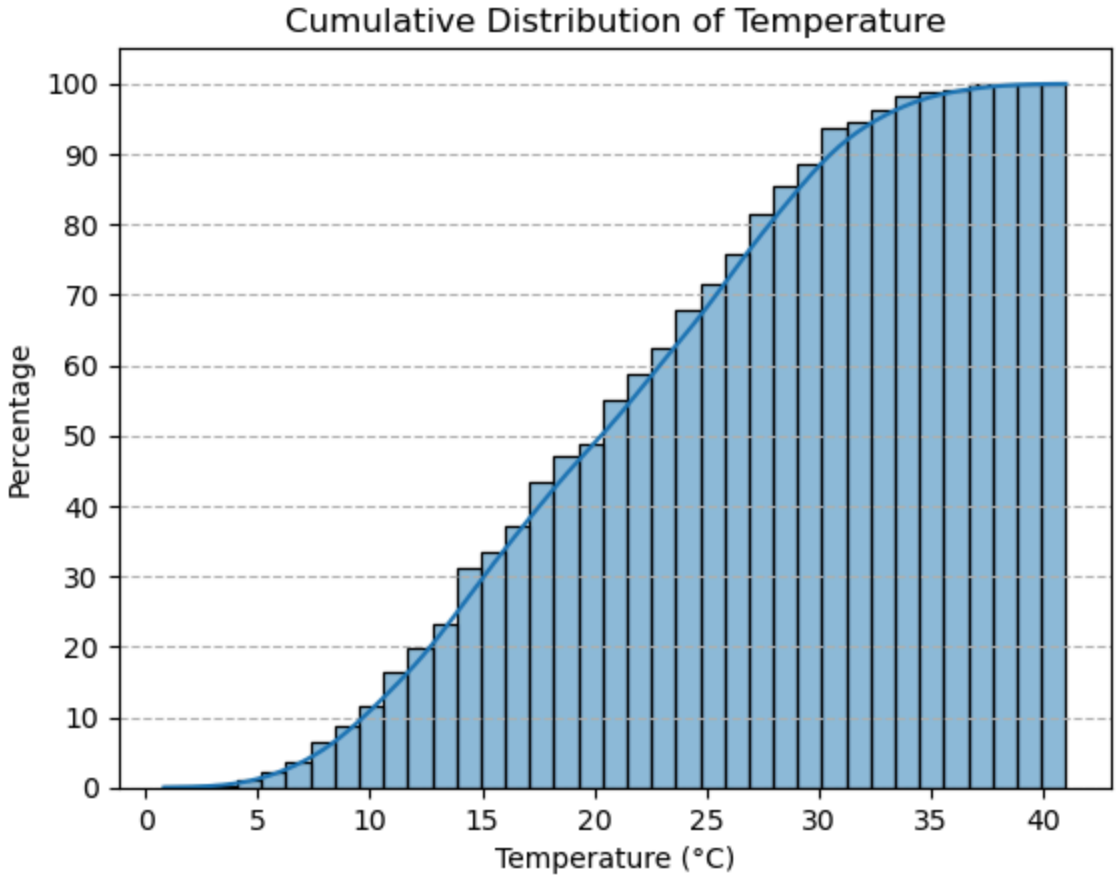
- Each count plot showcases the distribution of a specific categorical variable, such as seasons, holidays, working days, and weather conditions.
- By plotting the frequency of occurrence of each category within the variable, these count plots provide insights into the distribution patterns of the dataset.
- The count plots help in understanding the relative prevalence of different categories within each variable, facilitating exploratory data analysis.

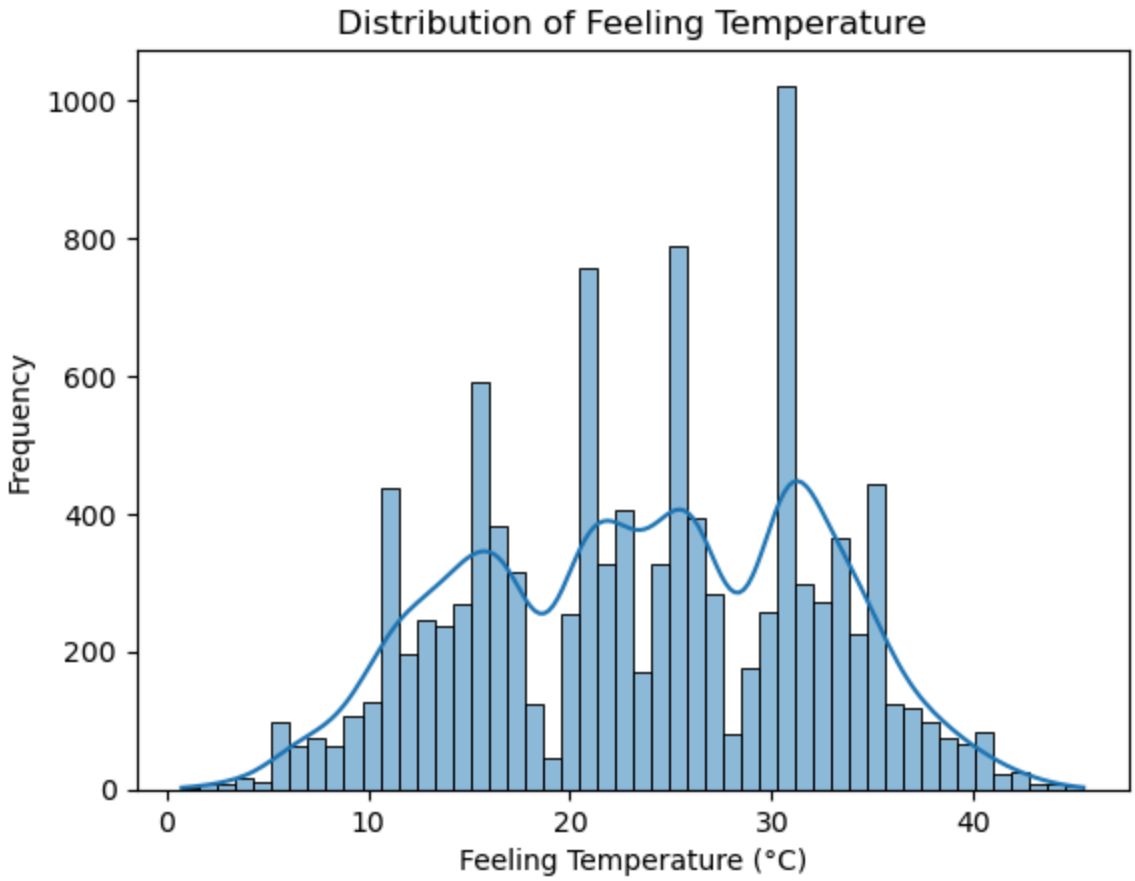
In [29]:
```
# Function to calculate mean and standard deviation
def calculate_statistics(feature):
    mean = np.round(df[feature].mean(), 2)
    std = np.round(df[feature].std(), 2)
    return mean, std

# Histogram plot for 'temp' feature with kernel density estimation
sns.histplot(data=df, x='temp', kde=True, bins=40)
plt.title('Distribution of Temperature')
plt.xlabel('Temperature (°C)')
plt.ylabel('Frequency')
plt.show()

# Calculating and displaying mean and standard deviation for 'temp' feature
temp_mean, temp_std = calculate_statistics('temp')
print(f"The mean and standard deviation of the temperature column are {temp_mean} and

# Cumulative distribution plot for 'temp' feature
sns.histplot(data=df, x='temp', kde=True, cumulative=True, stat='percent')
plt.title('Cumulative Distribution of Temperature')
plt.xlabel('Temperature (°C)')
plt.ylabel('Percentage')
plt.grid(axis='y', linestyle='--')
plt.yticks(np.arange(0, 101, 10))
plt.show()

# Histogram plot for 'atemp' feature with kernel density estimation
sns.histplot(data=df, x='atemp', kde=True, bins=50)
plt.title('Distribution of Feeling Temperature')
plt.xlabel('Feeling Temperature (°C)')
plt.ylabel('Frequency')
plt.show()

# Calculating and displaying mean and standard deviation for 'atemp' feature
atemp_mean, atemp_std = calculate_statistics('atemp')
print(f"The mean and standard deviation of the feeling temperature column are {atemp_m

# Histogram plot for 'humidity' feature with kernel density estimation
sns.histplot(data=df, x='humidity', kde=True, bins=50)
plt.title('Distribution of Humidity')
plt.xlabel('Humidity (%)')
plt.ylabel('Frequency')
plt.show()

# Calculating and displaying mean and standard deviation for 'humidity' feature
humidity_mean, humidity_std = calculate_statistics('humidity')
print(f"The mean and standard deviation of the humidity column are {humidity_mean} and

# Cumulative distribution plot for 'humidity' feature
sns.histplot(data=df, x='humidity', kde=True, cumulative=True, stat='percent')
```

```
plt.title('Cumulative Distribution of Humidity')
plt.xlabel('Humidity (%)')
plt.ylabel('Percentage')
plt.grid(axis='y', linestyle='--')
plt.yticks(np.arange(0, 101, 10))
plt.show()

# Histogram plot for 'windspeed' feature with kernel density estimation
sns.histplot(data=df, x='windspeed', kde=True, bins=50)
plt.title('Distribution of Windspeed')
plt.xlabel('Windspeed (km/h)')
plt.ylabel('Frequency')
plt.show()

# Histogram plot for 'casual' feature with kernel density estimation
sns.histplot(data=df, x='casual', kde=True, bins=50)
plt.title('Distribution of Casual Users')
plt.xlabel('Count of Casual Users')
plt.ylabel('Frequency')
plt.show()

# Cumulative distribution plot for 'casual' feature
sns.histplot(data=df, x='casual', kde=True, cumulative=True, stat='percent')
plt.title('Cumulative Distribution of Casual Users')
plt.xlabel('Count of Casual Users')
plt.ylabel('Percentage')
plt.grid(axis='y', linestyle='--')
plt.yticks(np.arange(0, 101, 10))
plt.show()

# Histogram plot for 'registered' feature with kernel density estimation
sns.histplot(data=df, x='registered', kde=True, bins=50)
plt.title('Distribution of Registered Users')
plt.xlabel('Count of Registered Users')
plt.ylabel('Frequency')
plt.show()

# Cumulative distribution plot for 'registered' feature
sns.histplot(data=df, x='registered', kde=True, cumulative=True, stat='percent')
plt.title('Cumulative Distribution of Registered Users')
plt.xlabel('Count of Registered Users')
plt.ylabel('Percentage')
plt.grid(axis='y', linestyle='--')
plt.yticks(np.arange(0, 101, 10))
plt.show()
```

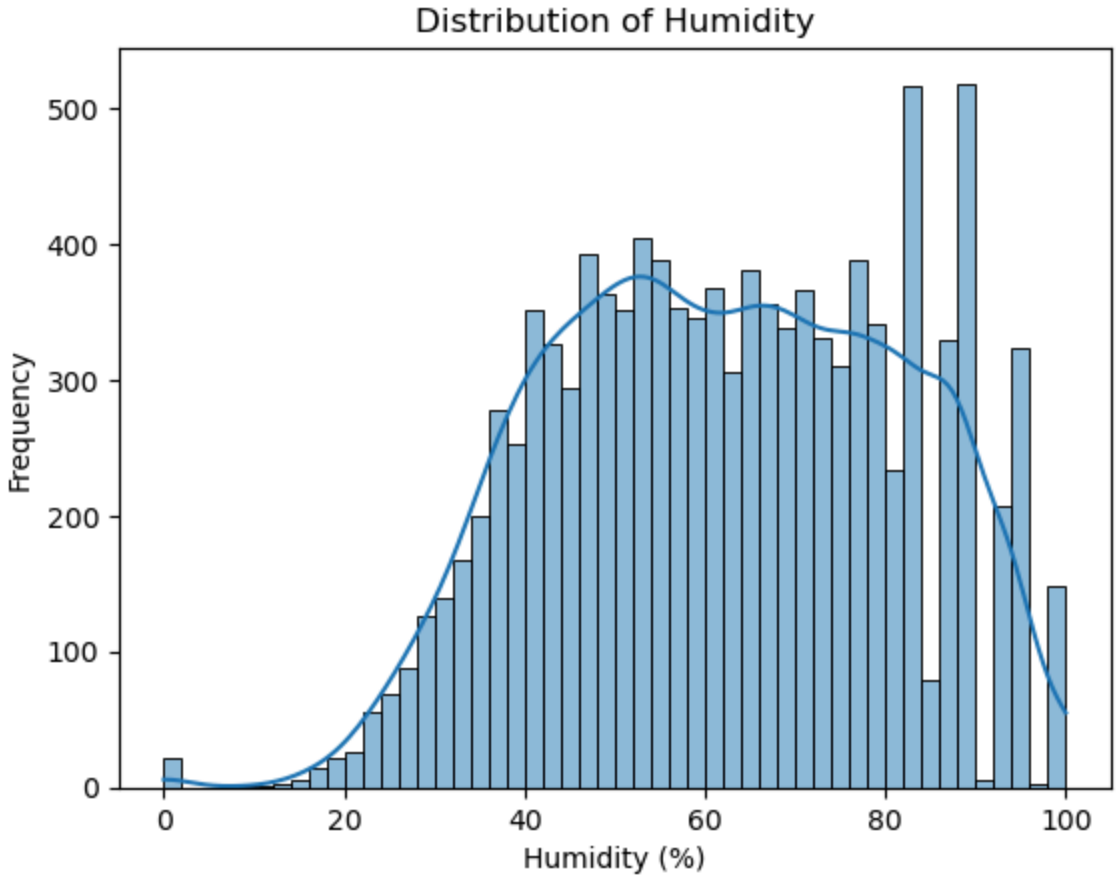


The mean and standard deviation of the temperature column are 20.229999542236328 and 7.79 respectively.

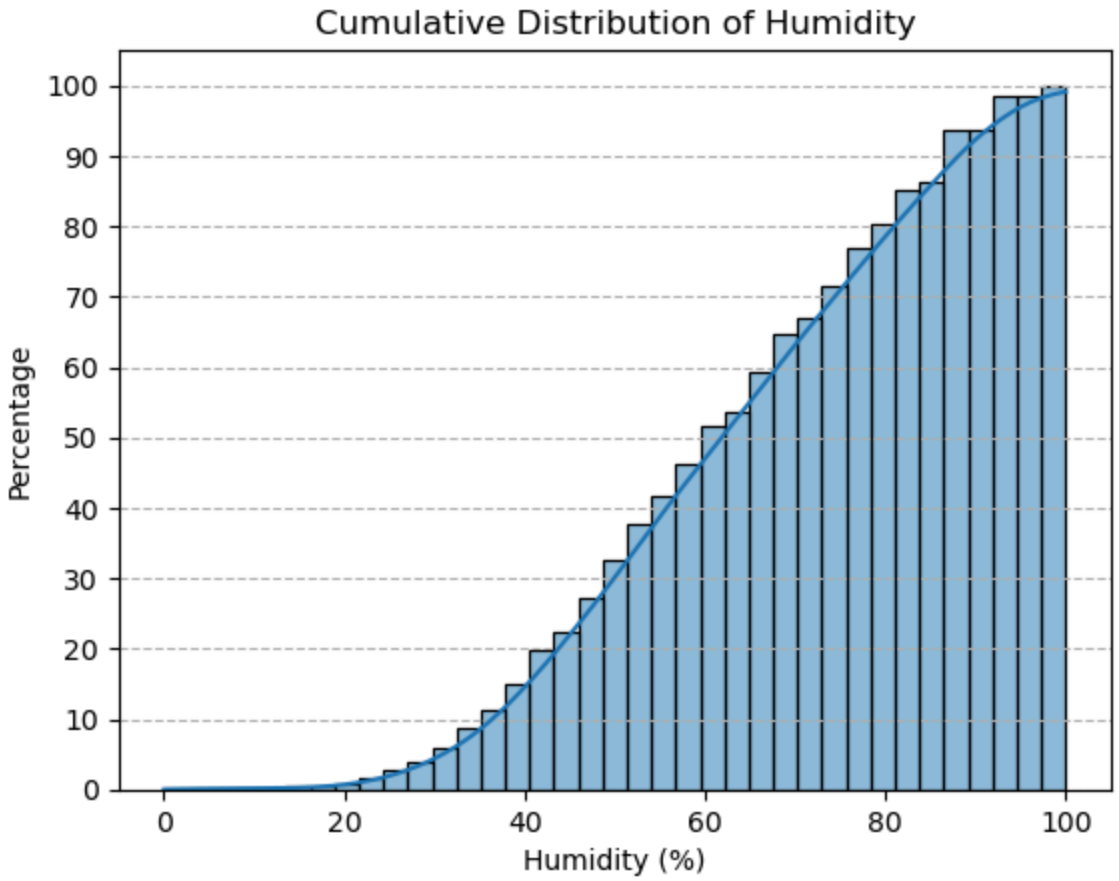


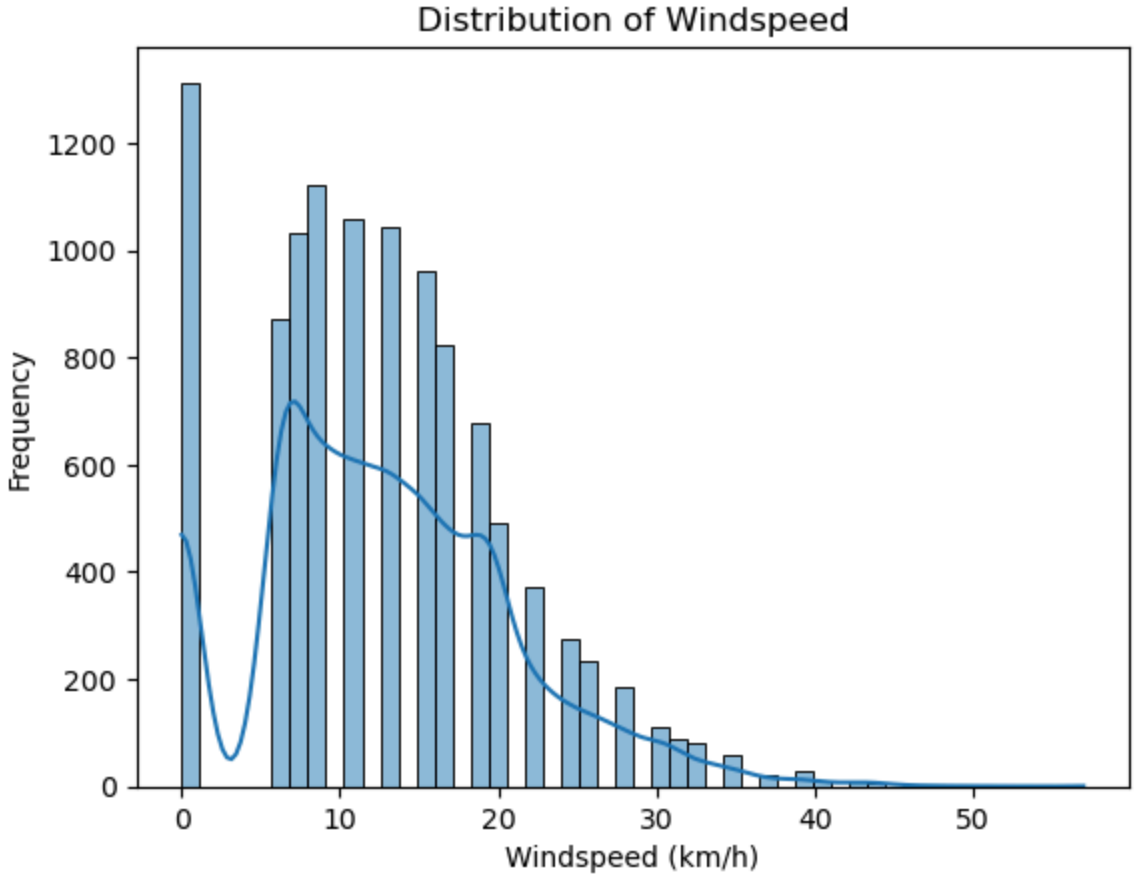


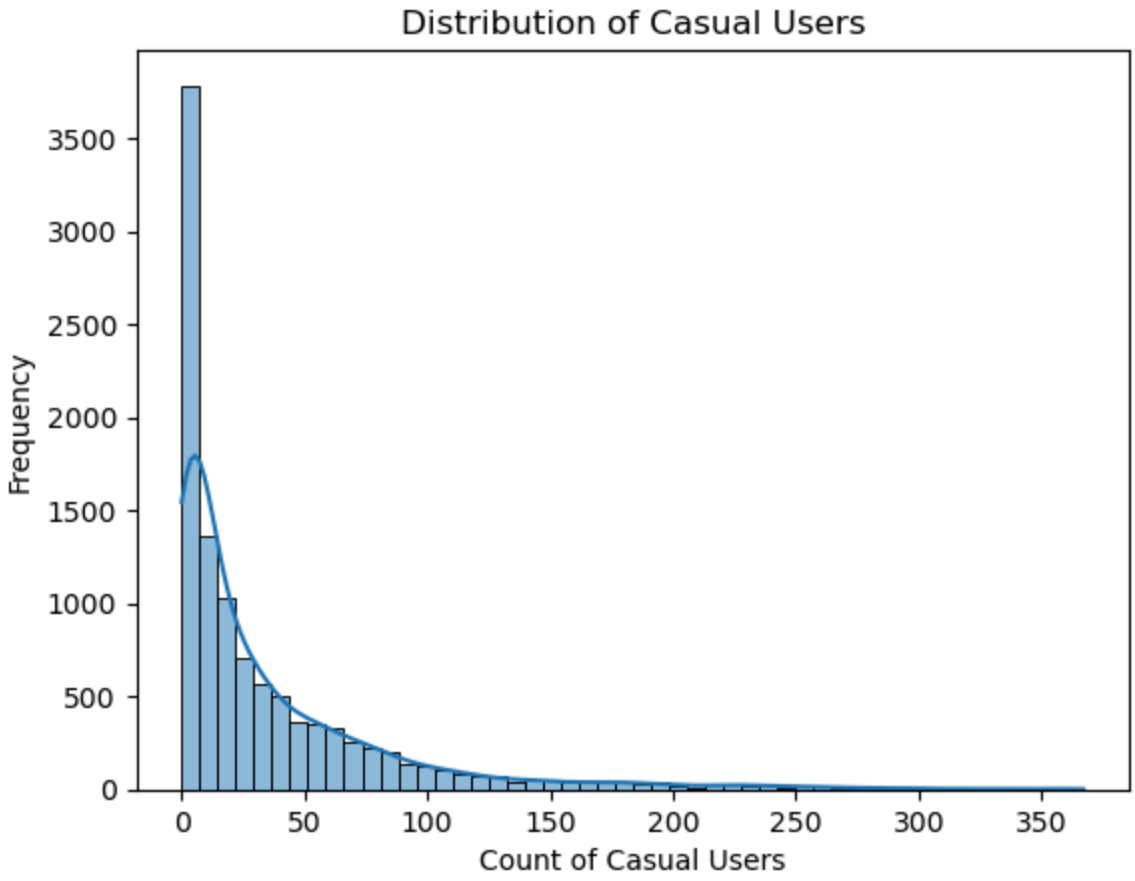
The mean and standard deviation of the feeling temperature column are 23.659999847412 11 and 8.47 respectively.

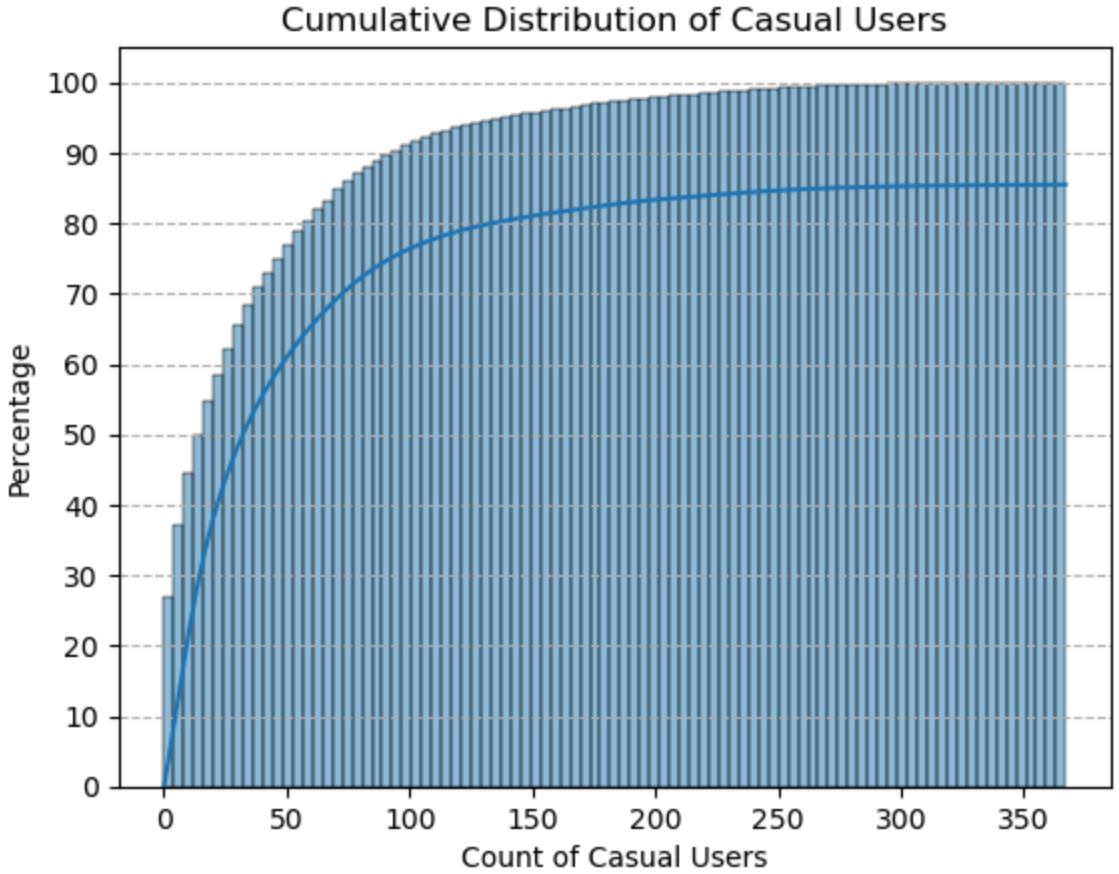


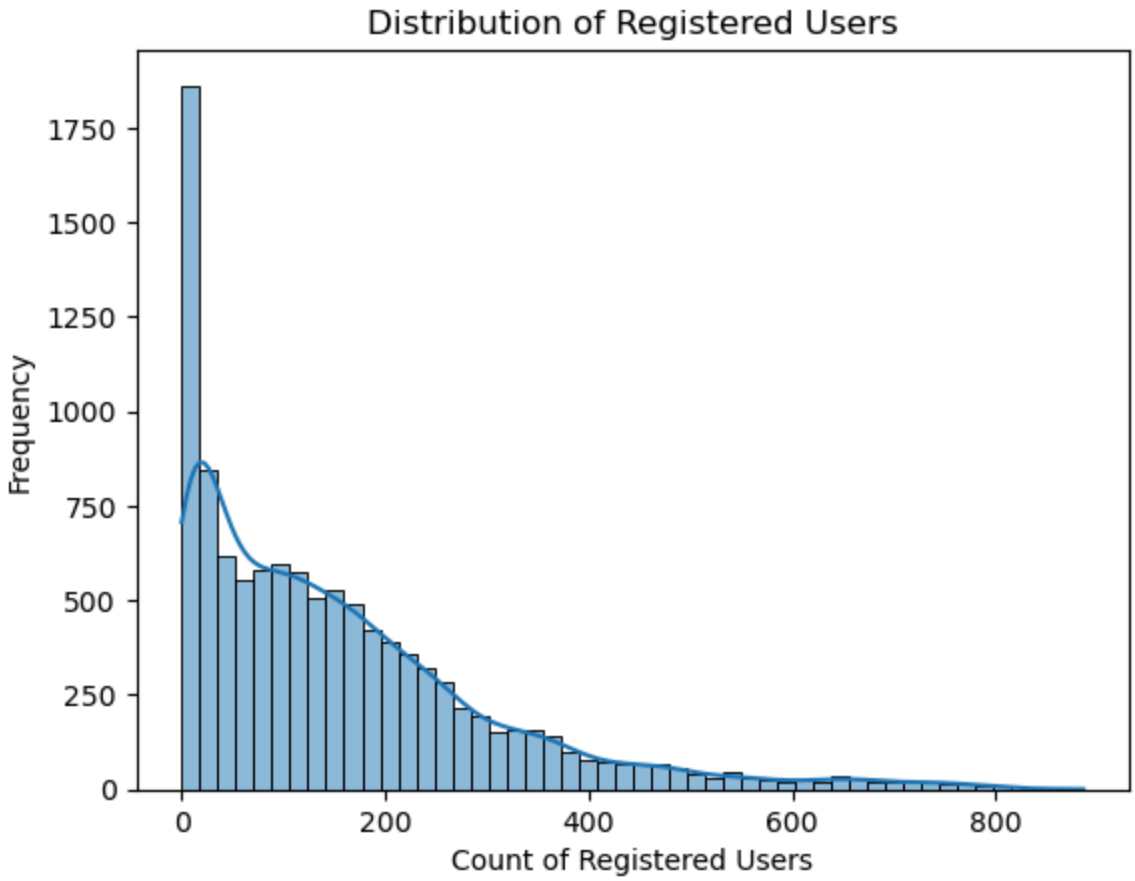
The mean and standard deviation of the humidity column are 61.89 and 19.25 respectively.

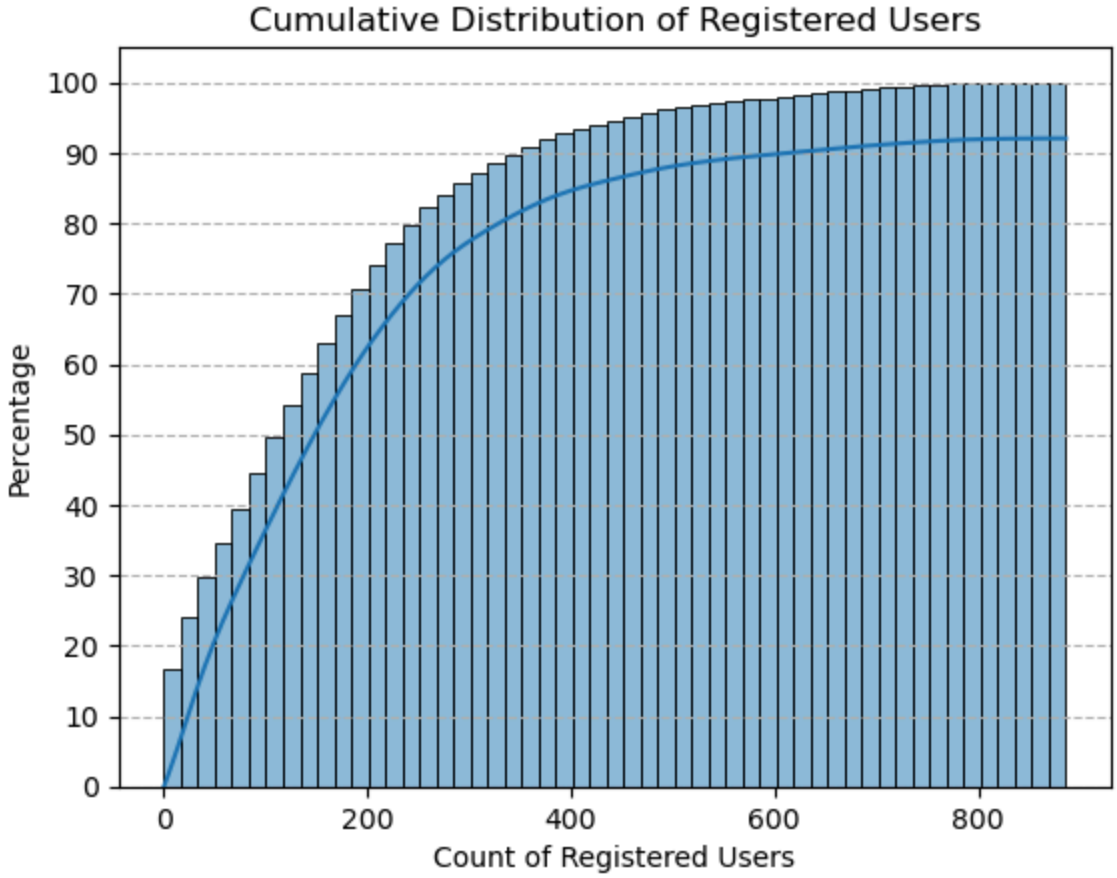

Cumulative Distribution of Humidity
Percentage
Humidity (%)
0
10
20
30
40
50
60
70
80
90
100
0
20
40
60
80
100


Distribution of Windspeed
Frequency
Windspeed (km/h)
0
200
400
600
800
1000
1200
0
10
20
30
40
50


Distribution of Casual Users
Frequency
Count of Casual Users
0
500
1000
1500
2000
2500
3000
3500
0
50
100
150
200
250
300
350


Cumulative Distribution of Casual Users
Percentage
Count of Casual Users
0
10
20
30
40
50
60
70
80
90
100
0
50
100
150
200
250
300
350

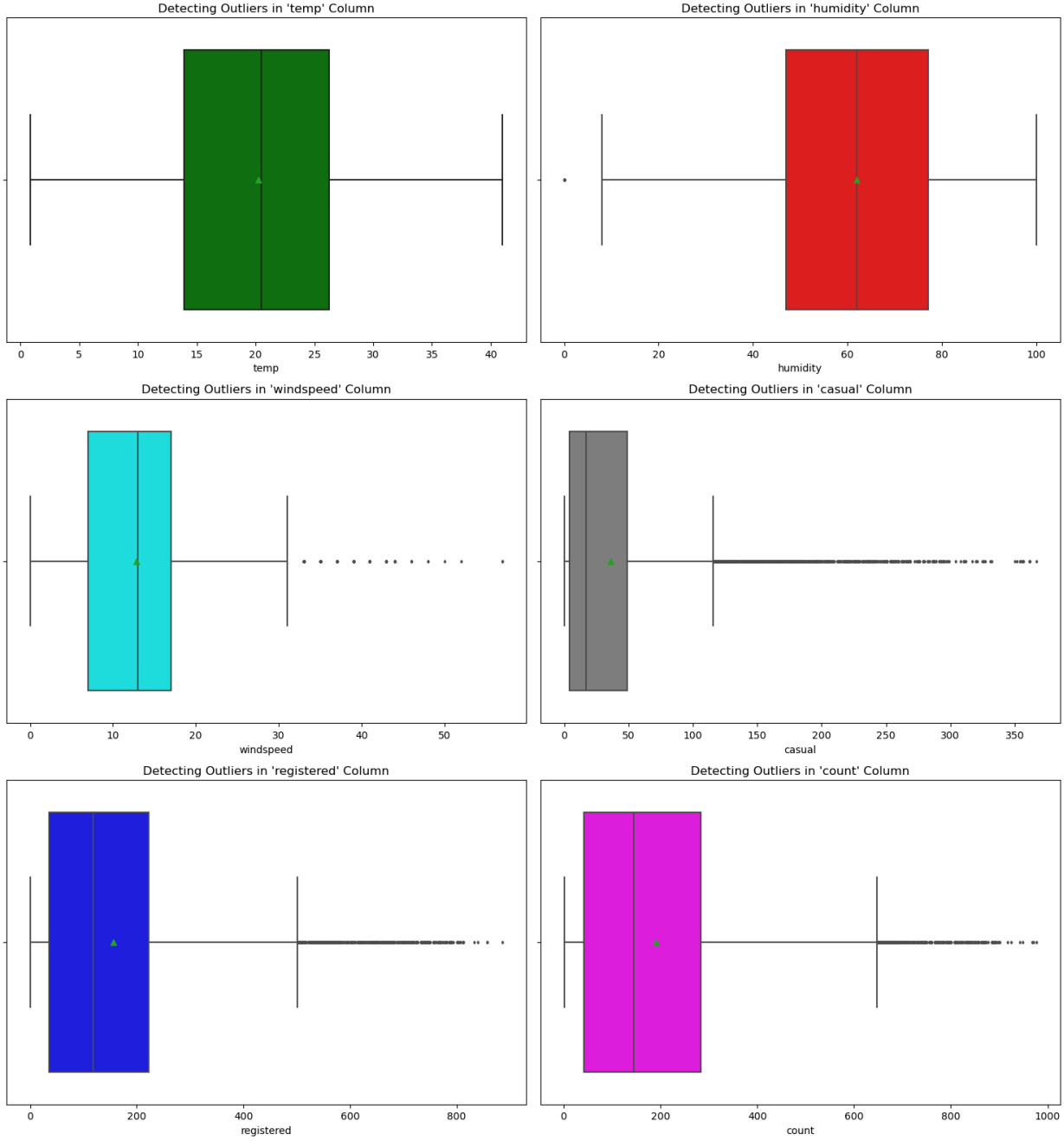

Distribution of Registered Users
Frequency
Count of Registered Users
0
250
500
750
1000
1250
1500
1750
0
200
400
600
800


Cumulative Distribution of Registered Users
Percentage
Count of Registered Users
0
10
20
30
40
50
60
70
80
90
100
0
200
400
600
800

- The code generates histogram plots for various features in the dataset to visualize their distributions.
- Each histogram plot includes a kernel density estimation (KDE) curve to represent the distribution shape more clearly.
- For the 'temp' feature, the mean and standard deviation are calculated and displayed.
- The cumulative distribution of 'temp' and 'humidity' features is plotted to show the percentage of values below certain thresholds.
- Similarly, histograms and cumulative distribution plots are generated for 'atemp', 'humidity', 'windspeed', 'casual', and 'registered' features.
- Insights are provided based on the cumulative distribution plots, highlighting the percentage of data falling within specific ranges for each feature.

Outliers Detection

In [30]:
```
columns = ['temp', 'humidity', 'windspeed', 'casual', 'registered', 'count']
colors = np.random.permutation(['red', 'blue', 'green', 'magenta', 'cyan', 'gray'])

plt.figure(figsize=(15, 16))

for idx, column in enumerate(columns, start=1):
    plt.subplot(3, 2, idx)
    plt.title(f"Detecting Outliers in '{column}' Column")
    sns.boxplot(data=df, x=df[column], color=colors[idx - 1], showmeans=True, fliersiz
    plt.plot()

plt.tight_layout()
plt.show()
```

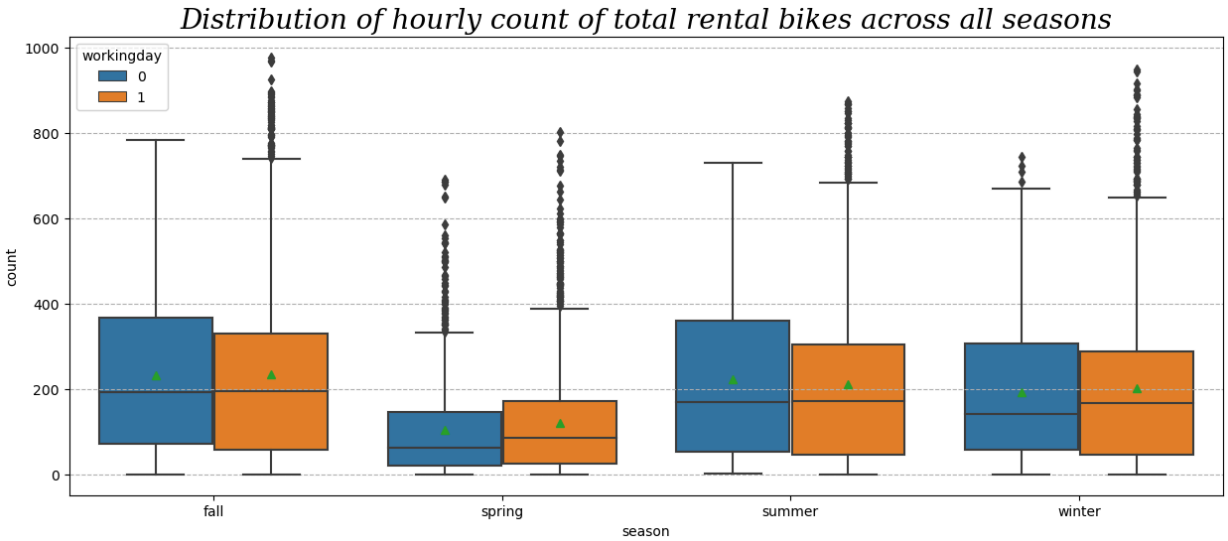


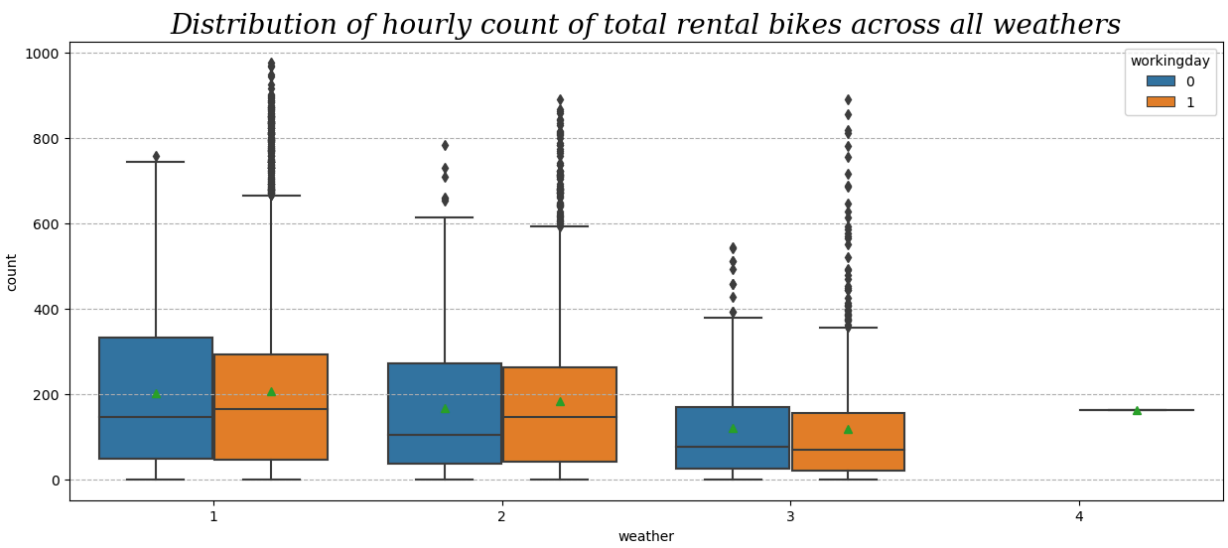
- The temperature column does not contain any outliers.
- Some outliers are present in the humidity column.
- Many outliers are present in the windspeed, casual, registered, and count columns.

Bivariate Analysis

In [31]:
```
# Boxplot showing the distribution of hourly count of total rental bikes across all se
plt.figure(figsize=(15, 6))
plt.title('Distribution of hourly count of total rental bikes across all seasons',
          fontdict={'size': 20, 'style': 'oblique', 'family': 'serif'})
sns.boxplot(data=df, x='season', y='count', hue='workingday', showmeans=True)
plt.grid(axis='y', linestyle='--')
plt.show()
```

```
# Interpretation: The hourly count of total rental bikes is higher in the fall season,
# It is generally low in the spring season.

# Boxplot showing the distribution of hourly count of total rental bikes across all we
plt.figure(figsize=(15, 6))
plt.title('Distribution of hourly count of total rental bikes across all weathers',
          fontdict={'size': 20, 'style': 'oblique', 'family': 'serif'})
sns.boxplot(data=df, x='weather', y='count', hue='workingday', showmeans=True)
plt.grid(axis='y', linestyle='--')
plt.show()
```





- The highest hourly count of total rental bikes occurs during the fall season, followed by summer and winter, while it tends to be lower during spring.
- Regarding weather conditions, the count is highest during clear and cloudy weather, followed by misty and rainy conditions.
  - Extreme weather conditions have fewer recorded instances.

Does the status of it being a working day influence the number of electric cycles rented?

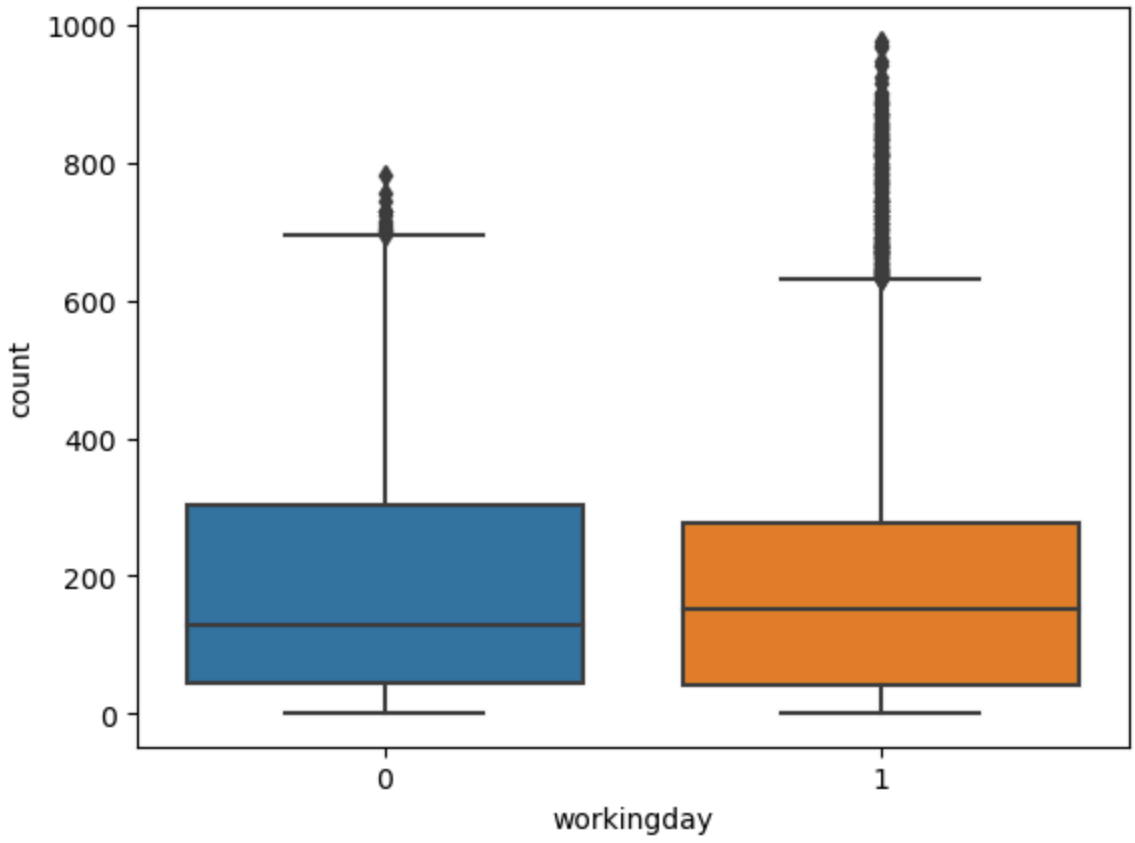
In [32]:
```
df.groupby(by = 'workingday')['count'].describe()
```

Out[32]:

workingday	count	mean	std	min	25%	50%	75%	max
0	3474.0	188.506621	173.724015	1.0	44.0	128.0	304.0	783.0
1	7412.0	193.011873	184.513659	1.0	41.0	151.0	277.0	977.0

In [33]:
```
sns.boxplot(data = df, x = 'workingday', y = 'count')
plt.plot()
```

Out[33]:
```
[]
```



Step 1: Set up Null Hypothesis

Null Hypothesis (H0): Working Day has no effect on the number of electric cycles rented.

Alternate Hypothesis (HA): Working Day has an effect on the number of electric cycles rented.

Step 2: Checking for basic assumptions for the hypothesis

- Distribution check using QQ Plot
- Homogeneity of Variances using Levene's test

Step 3: Define Test statistics; Distribution of T under H0.

If the assumptions of the T Test are met, we can proceed with performing the T Test for independent samples. Otherwise, we will perform the non-parametric test equivalent to the T Test for independent samples, i.e., Mann-Whitney U rank test for two independent samples.

Step 4: Compute the p-value and fix the value of alpha.

We set our alpha to be 0.05.

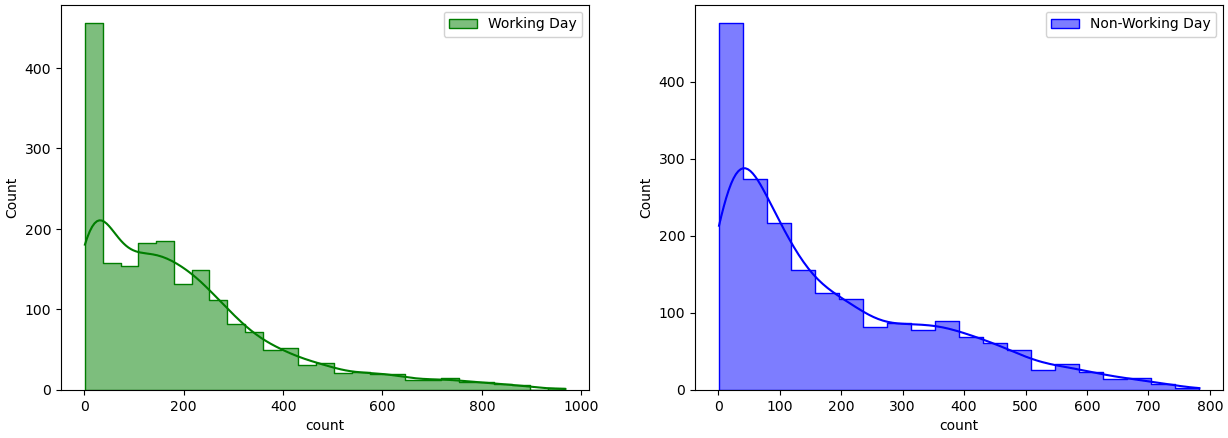
Step 5: Compare p-value and alpha.

- If p-value > alpha: Accept H0.
- If p-value < alpha: Reject H0.

Visual Tests to Check Normal Distribution of Samples

In [34]:
```
plt.figure(figsize=(15, 5))

# Plot for working days
plt.subplot(1, 2, 1)
sns.histplot(df.loc[df['workingday'] == 1, 'count'].sample(2000),
             element='step', color='green', kde=True, label='Working Day')
plt.legend()

# Plot for non-working days
plt.subplot(1, 2, 2)
sns.histplot(df.loc[df['workingday'] == 0, 'count'].sample(2000),
             element='step', color='blue', kde=True, label='Non-Working Day')
plt.legend()

plt.plot()
```

Out[34]:
```
[]
```



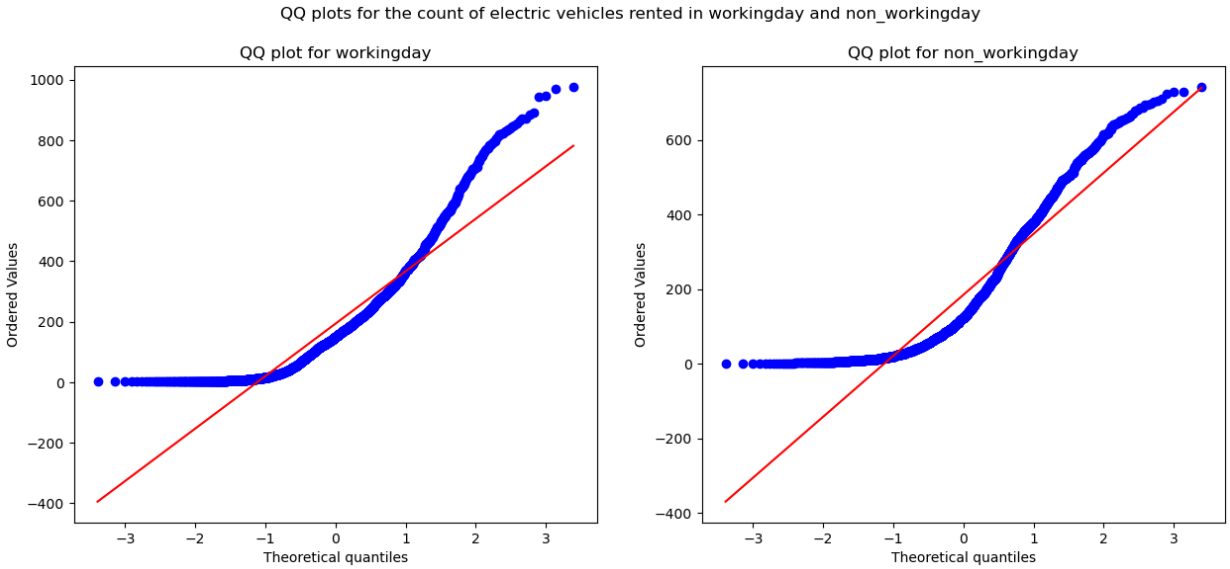
- The plots suggest that the distributions deviate from the normal distribution pattern.

Distribution check using QQ Plot

In [35]:
```
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for the count of electric vehicles rented in workingday and nor
stats.probplot(df.loc[df['workingday'] == 1, 'count'].sample(2000), plot=plt)
plt.title('QQ plot for workingday')
plt.subplot(1, 2, 2)
stats.probplot(df.loc[df['workingday'] == 0, 'count'].sample(2000), plot=plt)
```

```
plt.title('QQ plot for non_workingday')
plt.plot()
```

Out[35]: []



- It can be inferred from the above plots that the samples do not come from a normal distribution.

Applying the Shapiro-Wilk test for normality, the result indicates:

1.If the p-value is greater than 0.05, we accept that the sample follows a normal distribution.

2.If the p-value is less than 0.05, we reject the hypothesis that the sample follows a normal distribution.

With an alpha value of 0.05, we perform the Shapiro-Wilk test for normality.

Transform the data using the Box-Cox transformation and check if the transformed data follows a normal distribution.

In [36]:
```
test_stat, p_value = stats.shapiro(df.loc[df['holiday'] == 1, 'count'].sample(200))
print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

test_stat, p_value = stats.shapiro(df.loc[df['holiday'] == 0, 'count'].sample(200))
print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')
```

```
p-value 6.938977170634075e-11
The sample does not follow normal distribution
p-value 6.083708623677575e-12
The sample does not follow normal distribution
```

Transform the data using the Box-Cox transformation and check if the transformed data follows a normal distribution.

In [37]:
```
transformed_holiday = stats.boxcox(df.loc[df['holiday'] == 1, 'count'])[0]
test_stat, p_value = stats.shapiro(transformed_holiday)
print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

transformed_non_holiday = stats.boxcox(df.loc[df['holiday'] == 0, 'count'].sample(5000
test_stat, p_value = stats.shapiro(transformed_non_holiday)
print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')
```

```
p-value 2.1349180201468698e-07
The sample does not follow normal distribution
p-value 1.4035113358399901e-27
The sample does not follow normal distribution
```

- Even after applying the boxcox transformation to both the "holiday" and "non-holiday" datasets, the samples still do not conform to a normal distribution.

Homogeneity of Variances using Levene's test

In [38]:
```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = stats.levene(df.loc[df['holiday'] == 0, 'count'].sample(200),
                                  df.loc[df['holiday'] == 1, 'count'].sample(200))
print('p-value', p_value)
if p_value < 0.05:
    print('The samples do not have  Homogenous Variance')
else:
    print('The samples have Homogenous Variance ')
```

```
p-value 0.4951037821500369
The samples have Homogenous Variance
```

As the samples do not exhibit a normal distribution, we cannot apply the T-Test. Instead, we will conduct its non-parametric equivalent test, the Mann-Whitney U rank test for two independent samples.

In [39]:
```
# Ho : No.of electric cycles rented is similar for holidays and non-holidays
# Ha : No.of electric cycles rented is not similar for holidays and non-holidays days
# Assuming significance Level to be 0.05
```

```
# Test statistics : Mann-Whitney U rank test for two independent samples

test_stat, p_value = stats.mannwhitneyu(df.loc[df['holiday'] == 0, 'count'].sample(200
                                        df.loc[df['holiday'] == 1, 'count'].sample(200))
print('P-value :',p_value)
if p_value < 0.05:
    print('No.of electric cycles rented is not similar for holidays and non-holidays d
else:
    print('No.of electric cycles rented is similar for holidays and non-holidays')
```

```
P-value : 0.91012788470198
No.of electric cycles rented is similar for holidays and non-holidays
```

Hence, there is no statistically significant difference in the number of electric cycles rented between holidays and non-holidays.

Does weather vary depending on the season?

In [40]:
```
df[['weather', 'season']].describe()
```

Out[40]:

	weather	season
count	10886	10886
unique	4	4
top	1	winter
freq	7192	2734

- The statistical description above confirms that both the 'weather' and 'season' features are categorical.

STEP-1: Set up Null Hypothesis

Null Hypothesis (H0) - Weather is independent of season.

Alternate Hypothesis (HA) - Weather is dependent on seasons.

STEP-2: Define Test Statistics

Since we have two categorical features, the Chi-square test is applicable here. Under H0, the test statistic should follow the Chi-Square Distribution.

STEP-3: Checking for Basic Assumptions for the Hypothesis (Non-Parametric Test)

```
1. The data in the cells should be frequencies or counts of cases.
2. The levels (or categories) of the variables are mutually
exclusive. That is, a particular subject fits into one and    only
one level of each of the variables.
3. There are 2 variables, and both are measured as categories.
4. The expected value of each cell should be 5 or more in at least
80% of the cells, and no cell should have an expected value of less
```

than one.

STEP-4: Compute the p-value and Fix Value of Alpha

We will compute the Chi-square test p-value using the chi2_contingency function from scipy.stats. We set our alpha to be 0.05.

STEP-5: Compare p-value and Alpha

Based on the p-value, we will accept or reject H0.

- p-value > alpha: Accept H0
- p-value < alpha: Reject H0

The Chi-square statistic is a non-parametric (distribution-free) tool designed to analyze group differences when the dependent variable is measured at a nominal level. Like all non-parametric statistics, the Chi-square is robust with respect to the distribution of the data. Specifically, it does not require equality of variances among the study groups or homoscedasticity in the data.

In [41]:
```
# First, finding the contingency table such that each value is the total number of tot
  # for a particular season and weather
cross_table = pd.crosstab(index = df['season'],
                          columns = df['weather'],
                          values = df['count'],
                          aggfunc = np.sum).replace(np.nan, 0)
cross_table
```

Out[41]:

weather	1	2	3	4
season				
fall	470116	139386	31160	0
spring	223009	76406	12919	164
summer	426350	134177	27755	0
winter	356588	157191	30255	0

Since most cells in the above contingency table have counts of rented electric vehicles less than 5, we can remove the "weather 4" category and then proceed further.

In [42]:
```
cross_table = pd.crosstab(index = df['season'],
                          columns = df.loc[df['weather'] != 4, 'weather'],
                          values = df['count'],
                          aggfunc = np.sum).to_numpy()[:, :3]
cross_table
```

Out[42]:
```
array([[470116, 139386,  31160],
       [223009,  76406,  12919],
       [426350, 134177,  27755],
       [356588, 157191,  30255]], dtype=int64)
```

In [43]:
```
chi_test_stat, p_value, dof, expected = stats.chi2_contingency(observed = cross_table)
print('Test Statistic =', chi_test_stat)
print('p value =', p_value)
print('-' * 65)
print("Expected : '\n'", expected)
```

```
Test Statistic = 10838.372332480214
p value = 0.0
-----------------------------------------------------------------
Expected : '
' [[453484.88557396 155812.72247031  31364.39195574]
 [221081.86259035  75961.44434981  15290.69305984]
 [416408.3330293  143073.60199337  28800.06497733]
 [385087.91880639 132312.23118651  26633.8500071 ]]
```

Comparing p value with significance level

In [44]:
```
if p_value < 0.05:
    print('Reject the Null Hypothesis')
else:
    print('Failed to reject the Null Hypothesis')
```

```
Reject the Null Hypothesis
```

Hence, there exists a statistically significant relationship between weather and season, as indicated by the number of bikes rented.

Is the number of rented cycles similar or different across different weather conditions?

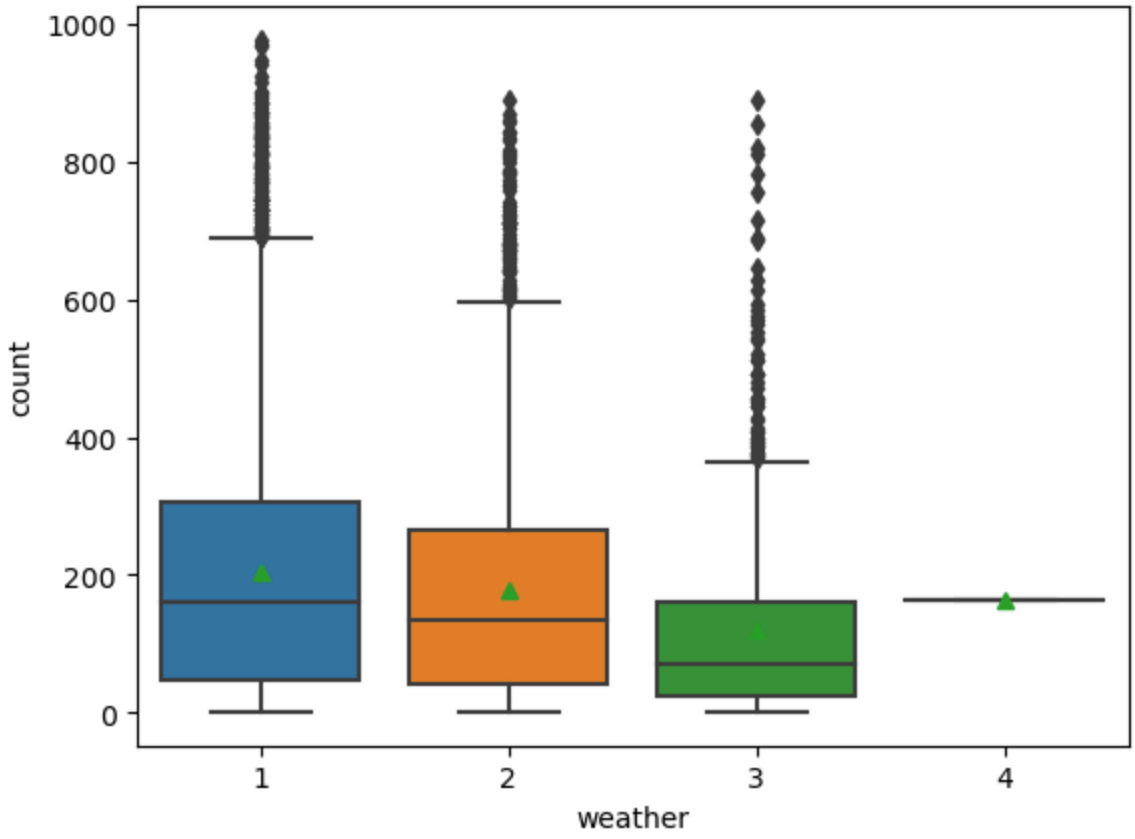
In [45]:
```
df.groupby(by = 'weather')['count'].describe()
```

Out[45]:

weather	count	mean	std	min	25%	50%	75%	max
1	7192.0	205.236791	187.959566	1.0	48.0	161.0	305.0	977.0
2	2834.0	178.955540	168.366413	1.0	41.0	134.0	264.0	890.0
3	859.0	118.846333	138.581297	1.0	23.0	71.0	161.0	891.0
4	1.0	164.000000	NaN	164.0	164.0	164.0	164.0	164.0

In [46]:
```
sns.boxplot(data = df, x = 'weather', y = 'count', showmeans = True)
plt.plot()
```

Out[46]:
```
[]
```



In [47]:
```
df_weather1 = df.loc[df['weather'] == 1]
df_weather2 = df.loc[df['weather'] == 2]
df_weather3 = df.loc[df['weather'] == 3]
df_weather4 = df.loc[df['weather'] == 4]
len(df_weather1), len(df_weather2), len(df_weather3), len(df_weather4)
```

Out[47]:
```
(7192, 2834, 859, 1)
```

STEP-1: Setting up the Null Hypothesis

Null Hypothesis (H0): The mean number of cycles rented per hour is the same for weather conditions 1, 2, and 3.

Alternate Hypothesis (HA): The mean number of cycles rented per hour is different for at least one weather condition among 1, 2, and 3.

STEP-2: Checking the Basic Assumptions for the Hypothesis

```
1. Normality check using QQ Plot. If the distribution is not normal,
use the Box-Cox transform to normalize it.
2. Homogeneity of Variances using Levene's test.
3. Ensuring each observation is independent.
```

STEP-3: Defining Test Statistics

The test statistic for a One-Way ANOVA is denoted as F. For an independent variable with k groups, the F statistic evaluates whether the group means are significantly different.

F = MSB / MSW

Under H0, the test statistic should follow the F-Distribution.

STEP-4: Deciding the Type of Test

We will be performing a right-tailed F-test.

STEP-5: Computing the p-value and Setting the Value of Alpha

We will compute the p-value using the ANOVA test and set our alpha to be 0.05.

STEP-6: Comparing the p-value and Alpha

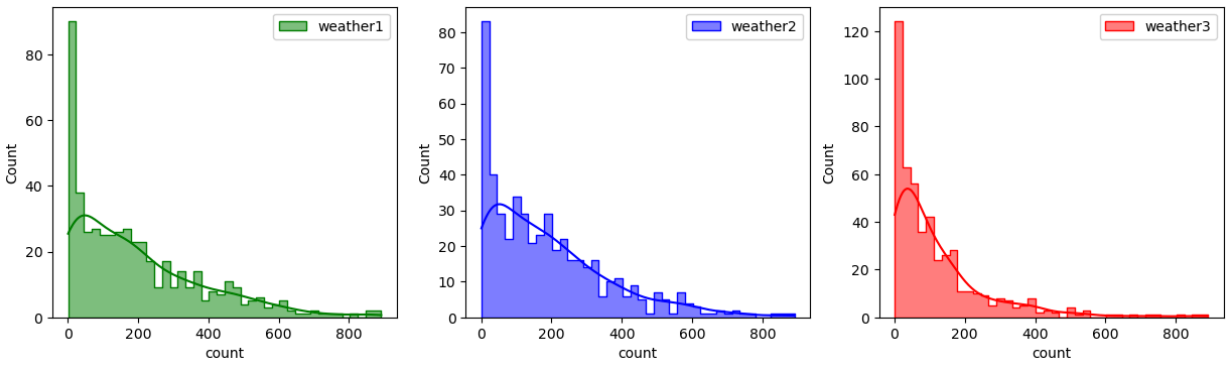
Based on the p-value:

- If p-val > alpha: Accept H0
- If p-val < alpha: Reject H0

Visual Tests to know if the samples follow normal distribution

In [48]:
```
plt.figure(figsize = (15, 4))
plt.subplot(1, 3, 1)
sns.histplot(df_weather1.loc[:, 'count'].sample(500), bins = 40,
             element = 'step', color = 'green', kde = True, label = 'weather1')
plt.legend()
plt.subplot(1, 3, 2)
sns.histplot(df_weather2.loc[:, 'count'].sample(500), bins = 40,
             element = 'step', color = 'blue', kde = True, label = 'weather2')
plt.legend()
plt.subplot(1, 3, 3)
sns.histplot(df_weather3.loc[:, 'count'].sample(500), bins = 40,
             element = 'step', color = 'red', kde = True, label = 'weather3')
plt.legend()
plt.plot()
```

Out[48]:
```
[]
```

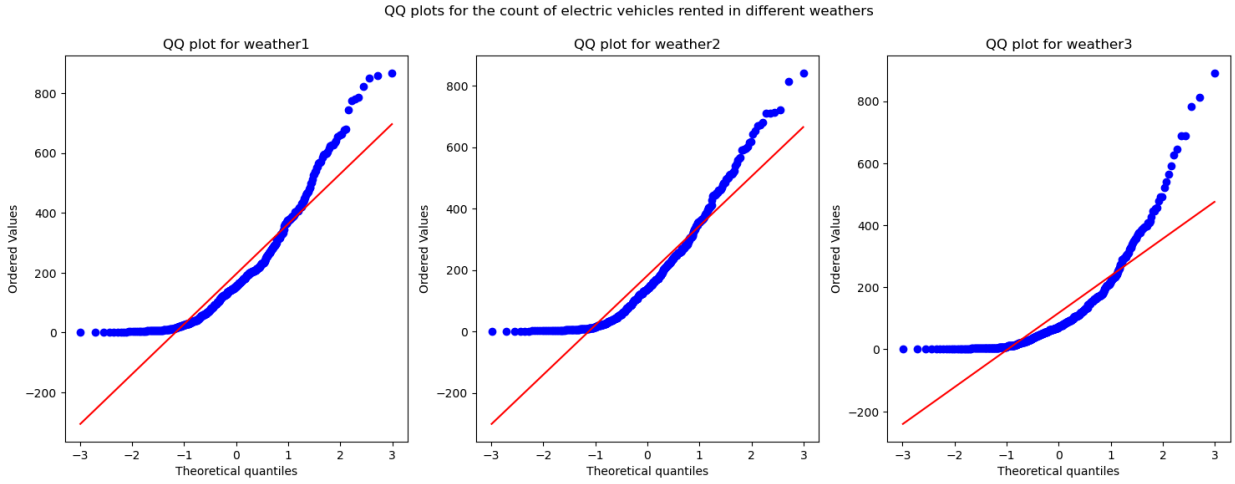


- It can be observed from the plot above that the distributions do not adhere to the normal distribution assumption.

Performing a distribution check using the QQ Plot.

In [49]:
```
plt.figure(figsize = (18, 6))
plt.subplot(1, 3, 1)
plt.suptitle('QQ plots for the count of electric vehicles rented in different weathers
stats.probplot(df_weather1.loc[:, 'count'].sample(500), plot = plt, dist = 'norm')
plt.title('QQ plot for weather1')
plt.subplot(1, 3, 2)
stats.probplot(df_weather2.loc[:, 'count'].sample(500), plot = plt, dist = 'norm')
plt.title('QQ plot for weather2')
plt.subplot(1, 3, 3)
stats.probplot(df_weather3.loc[:, 'count'].sample(500), plot = plt, dist = 'norm')
plt.title('QQ plot for weather3')
plt.plot()
```

Out[49]:
```
[]
```



- It is evident from the plots above that the samples deviate from a normal distribution.

Proceeding to apply the Shapiro-Wilk test for normality:

- If the p-value is less than 0.05, the sample does not follow a normal distribution.
- If the p-value is greater than or equal to 0.05, the sample follows a normal distribution.

In [50]:
```
# Shapiro-Wilk test for normality
test_stat, p_value = stats.shapiro(df_weather1.loc[:, 'count'].sample(500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
else:
    print('The sample follows a normal distribution')

test_stat, p_value = stats.shapiro(df_weather2.loc[:, 'count'].sample(500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
```

```
else:
    print('The sample follows a normal distribution')

test_stat, p_value = stats.shapiro(df_weather3.loc[:, 'count'].sample(500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
else:
    print('The sample follows a normal distribution')

# Transforming the data using Box-Cox transformation and checking if the transformed d

transformed_weather1 = stats.boxcox(df_weather1.loc[:, 'count'].sample(5000))[0]
test_stat, p_value = stats.shapiro(transformed_weather1)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
else:
    print('The sample follows a normal distribution')

transformed_weather2 = stats.boxcox(df_weather2.loc[:, 'count'])[0]
test_stat, p_value = stats.shapiro(transformed_weather2)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
else:
    print('The sample follows a normal distribution')

transformed_weather3 = stats.boxcox(df_weather3.loc[:, 'count'])[0]
test_stat, p_value = stats.shapiro(transformed_weather3)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow a normal distribution')
else:
    print('The sample follows a normal distribution')

# Even after applying the Box-Cox transformation on each of the weather data, the samp

# Homogeneity of Variances using Levene's test

# Null Hypothesis (H0) - Homogeneous Variance

# Alternate Hypothesis (HA) - Non-Homogeneous Variance

test_stat, p_value = stats.levene(df_weather1.loc[:, 'count'].sample(500),
                                  df_weather2.loc[:, 'count'].sample(500),
                                  df_weather3.loc[:, 'count'].sample(500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The samples do not have Homogeneous Variance')
else:
    print('The samples have Homogeneous Variance')
```

```
p-value: 1.594059112216086e-19
The sample does not follow a normal distribution
p-value: 5.671198669729392e-18
The sample does not follow a normal distribution
p-value: 4.965126036201783e-25
The sample does not follow a normal distribution
p-value: 1.811960728812188e-27
The sample does not follow a normal distribution
p-value: 1.9219748327822736e-19
The sample does not follow a normal distribution
p-value: 1.4137293646854232e-06
The sample does not follow a normal distribution
p-value: 8.809975190191603e-14
The samples do not have Homogeneous Variance
```

- Since the samples are not normally distributed and do not have the same variance, f_oneway test cannot be performed here, we can perform its non parametric equivalent test i.e., Kruskal-Wallis H-test for independent samples.

In [51]:
```
# Null Hypothesis (H0): Mean number of cycles rented is the same for different weather
# Alternative Hypothesis (HA): Mean number of cycles rented is different for different
# Assuming a significance level of 0.05
alpha = 0.05

test_stat, p_value = stats.kruskal(df_weather1, df_weather2, df_weather3)
print('Test Statistic =', test_stat)
print('p value =', p_value)
```

```
Test Statistic = [1.36471292e+01 3.87838808e+01 5.37649760e+00 1.56915686e+01
 1.08840000e+04 3.70017441e+01 4.14298489e+01 1.83168690e+03
 2.80380482e+01 2.84639685e+02 1.73745440e+02 2.04955668e+02
 7.08445555e+01]
p value = [1.08783632e-03 3.78605818e-09 6.79999165e-02 3.91398508e-04
 0.00000000e+00 9.22939752e-09 1.00837627e-09 0.00000000e+00
 8.15859150e-07 1.55338046e-62 1.86920588e-38 3.12206618e-45
 4.13333147e-16]
```

Comparing p value with significance level

In [52]:
```
if any(p_value < alpha):
    print('Reject the Null Hypothesis')
else:
    print('Failed to reject the Null Hypothesis')
```

```
Reject the Null Hypothesis
```

Therefore, the average number of rental bikes is statistically different for different weathers.

Is the number of cycles rented is similar or different in different season ?

In [53]:
```
df.groupby(by = 'season')['count'].describe()
```

Out[53]:

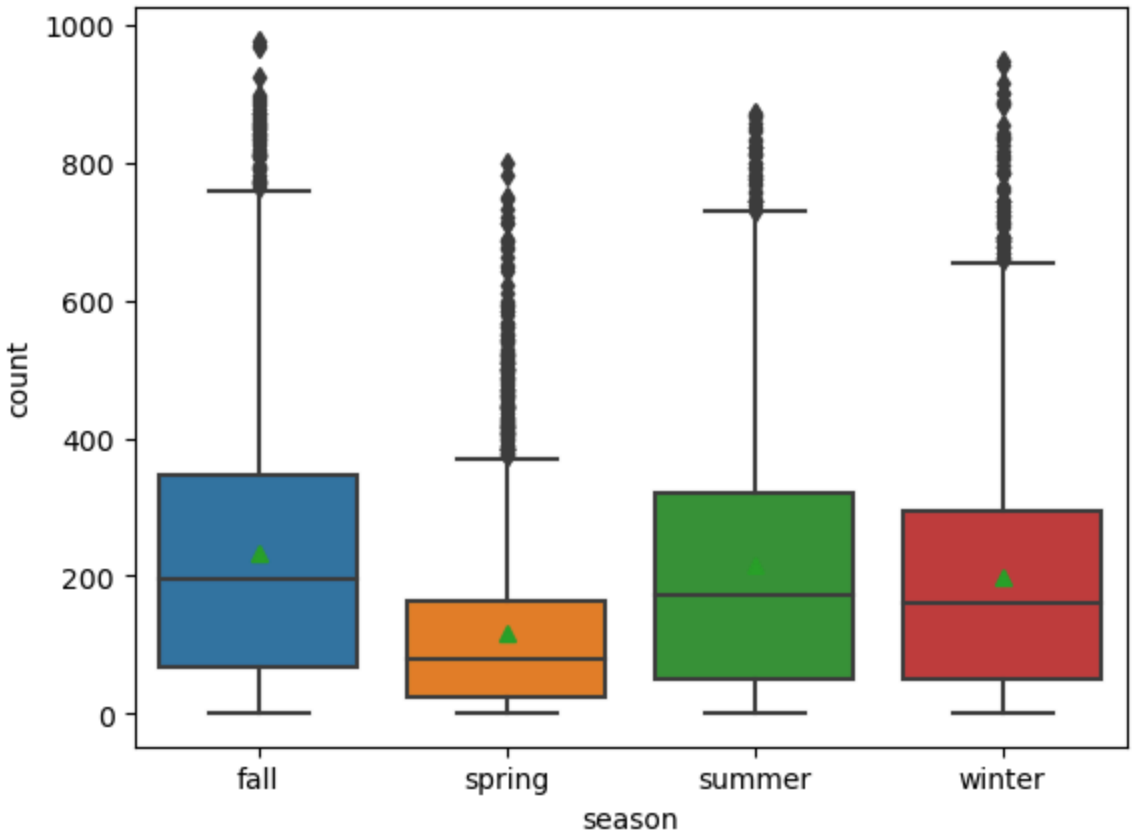
season	count	mean	std	min	25%	50%	75%	max
fall	2733.0	234.417124	197.151001	1.0	68.0	195.0	347.0	977.0
spring	2686.0	116.343261	125.273974	1.0	24.0	78.0	164.0	801.0
summer	2733.0	215.251372	192.007843	1.0	49.0	172.0	321.0	873.0
winter	2734.0	198.988296	177.622409	1.0	51.0	161.0	294.0	948.0

In [54]:
```
df_season_spring = df.loc[df['season'] == 'spring', 'count']
df_season_summer = df.loc[df['season'] == 'summer', 'count']
df_season_fall = df.loc[df['season'] == 'fall', 'count']
df_season_winter = df.loc[df['season'] == 'winter', 'count']
len(df_season_spring), len(df_season_summer), len(df_season_fall), len(df_season_winte
```

Out[54]:
```
(2686, 2733, 2733, 2734)
```

In [55]:
```
sns.boxplot(data = df, x = 'season', y = 'count', showmeans = True)
plt.plot()
```

Out[55]:
```
[]
```



STEP-1: Set up Null Hypothesis

Null Hypothesis (H0): The mean number of cycles rented per hour is the same for all seasons.

Alternate Hypothesis (HA): The mean number of cycles rented per hour is different for at least one season.

STEP-2: Checking for basic assumptions for the hypothesis

1. Normality check using QQ Plot. If the distribution is not normal, use Box-Cox transformation to transform it to a normal distribution.
2. Homogeneity of Variances using Levene's test.
3. Each observation is independent.

STEP-3: Define Test Statistics

The test statistic for a One-Way ANOVA is denoted as F. For an independent variable with k groups, the F statistic evaluates whether the group means are significantly different.

[F = \frac{MSB}{MSW}]

Under H0, the test statistic should follow the F-Distribution.

STEP-4: Decide the kind of test

We will be performing a right-tailed F-test.

STEP-5: Compute the p-value and fix value of alpha

We will compute the ANOVA test p-value using the `f_oneway` function from `scipy.stats`. We set our alpha to be 0.05.

STEP-6: Compare p-value and alpha

Based on the p-value, we will accept or reject H0.

- If p-value > alpha: Accept H0
- If p-value < alpha: Reject H0

The one-way ANOVA compares the means between the groups you are interested in and determines whether any of those means are statistically significantly different from each other.

Specifically, it tests the null hypothesis (H0):

[\mu_1 = \mu_2 = \mu_3 = \ldots = \mu_k]

where:

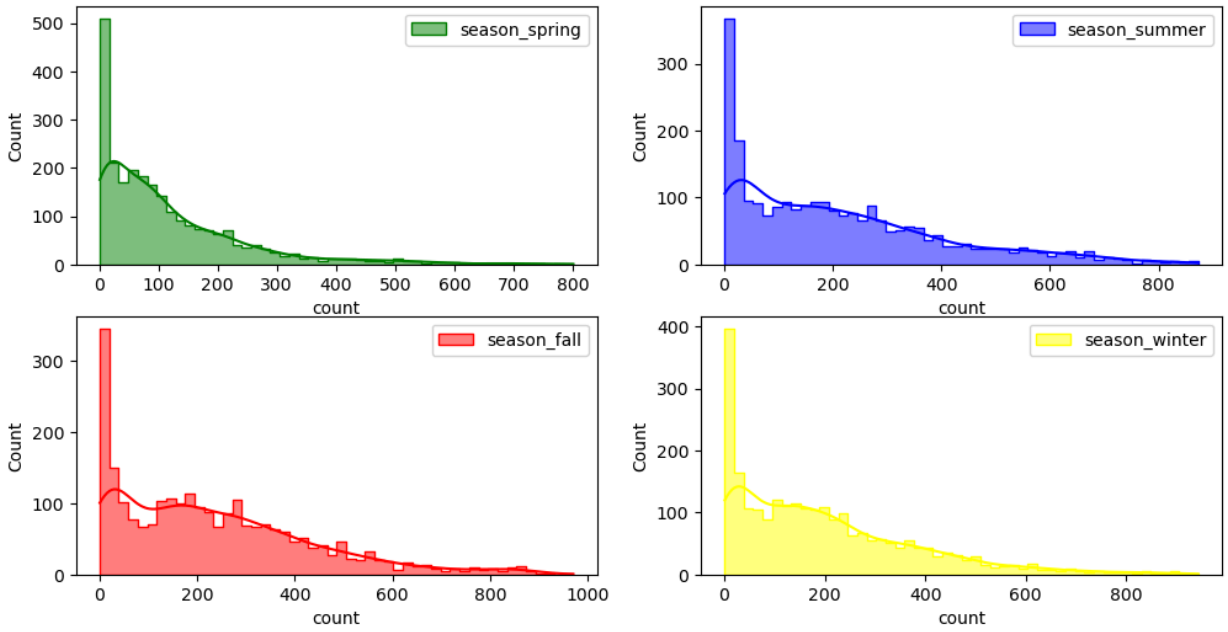
- (\mu) = group mean
- (k) = number of groups

If, however, the one-way ANOVA returns a statistically significant result, we accept the alternative hypothesis (HA), which is that there are at least two group means that are statistically significantly different from each other.

Visual Tests to know if the samples follow normal distribution

In [56]:
```
plt.figure(figsize = (12, 6))
plt.subplot(2, 2, 1)
sns.histplot(df_season_spring.sample(2500), bins = 50,
             element = 'step', color = 'green', kde = True, label = 'season_spring')
plt.legend()
plt.subplot(2, 2, 2)
sns.histplot(df_season_summer.sample(2500), bins = 50,
             element = 'step', color = 'blue', kde = True, label = 'season_summer')
plt.legend()
plt.subplot(2, 2, 3)
sns.histplot(df_season_fall.sample(2500), bins = 50,
             element = 'step', color = 'red', kde = True, label = 'season_fall')
plt.legend()
plt.subplot(2, 2, 4)
sns.histplot(df_season_winter.sample(2500), bins = 50,
             element = 'step', color = 'yellow', kde = True, label = 'season_winter')
plt.legend()
plt.plot()
```

Out[56]:
```
[]
```



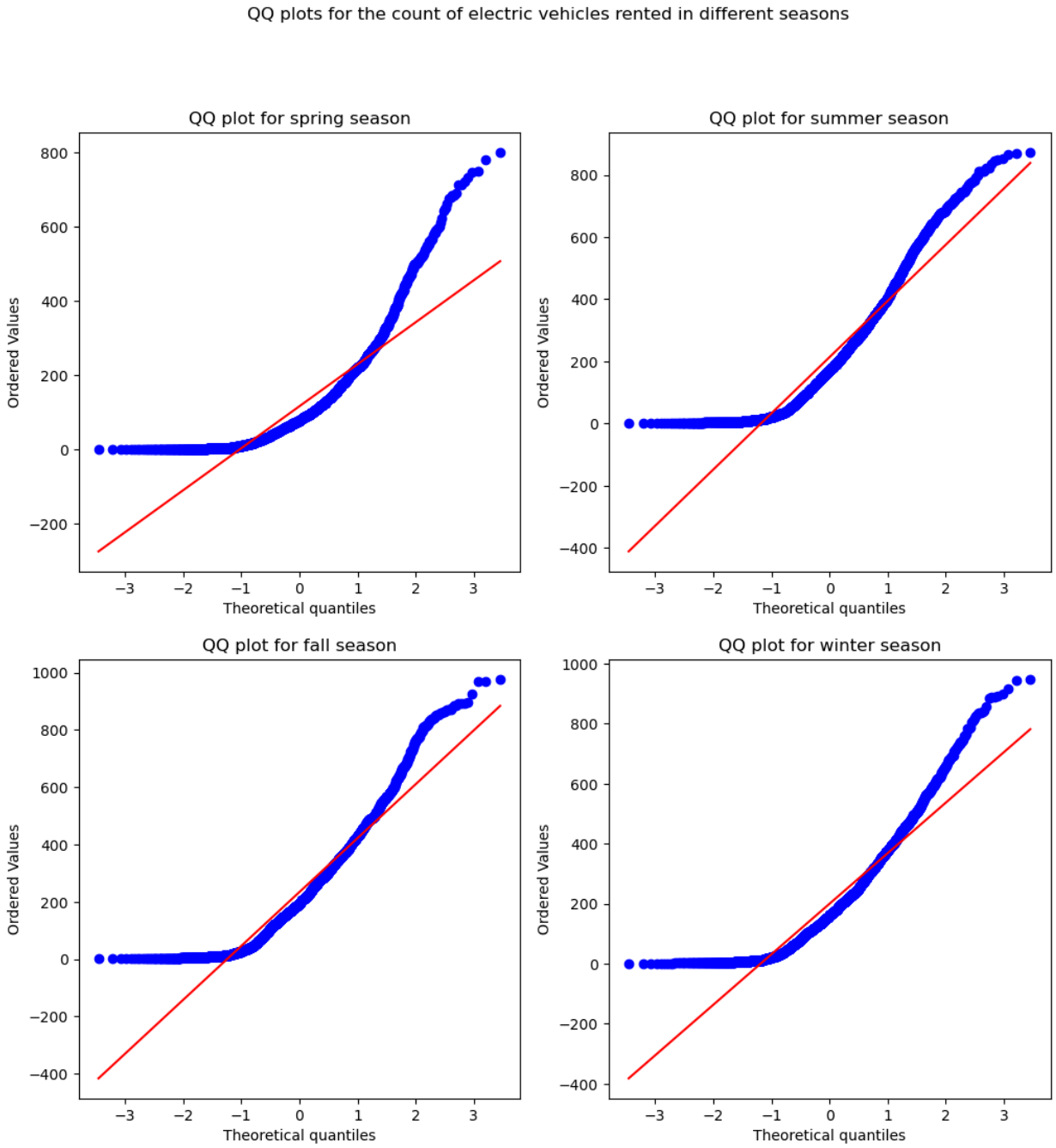
- The plots above indicate that the distributions do not adhere to a normal distribution pattern.

Checking the distribution using QQ plots.

In [57]:
```
plt.figure(figsize = (12, 12))
plt.subplot(2, 2, 1)
plt.suptitle('QQ plots for the count of electric vehicles rented in different seasons')
stats.probplot(df_season_spring.sample(2500), plot = plt, dist = 'norm')
plt.title('QQ plot for spring season')

plt.subplot(2, 2, 2)
stats.probplot(df_season_summer.sample(2500), plot = plt, dist = 'norm')
plt.title('QQ plot for summer season')
```

```
plt.subplot(2, 2, 3)
stats.probplot(df_season_fall.sample(2500), plot = plt, dist = 'norm')
plt.title('QQ plot for fall season')

plt.subplot(2, 2, 4)
stats.probplot(df_season_winter.sample(2500), plot = plt, dist = 'norm')
plt.title('QQ plot for winter season')
plt.plot()
```

Out[57]: []



- It can be inferred from the above plots that the distributions do not follow a normal distribution.
- It is evident from the plots that the samples do not adhere to a normal distribution.

To confirm this observation, the Shapiro-Wilk test for normality is applied, where:

- If the p-value is less than 0.05, the sample does not follow a normal distribution.
- If the p-value is greater than or equal to 0.05, the sample follows a normal distribution.

The significance level (alpha) is set to 0.05.

In [58]:
```
# Checking normality using Shapiro-Wilk test
test_stat, p_value = stats.shapiro(df_season_spring.sample(2500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

test_stat, p_value = stats.shapiro(df_season_summer.sample(2500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

test_stat, p_value = stats.shapiro(df_season_fall.sample(2500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

test_stat, p_value = stats.shapiro(df_season_winter.sample(2500))
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

# Transforming the data using Box-Cox transformation
transformed_df_season_spring = stats.boxcox(df_season_spring.sample(2500))[0]
test_stat, p_value = stats.shapiro(transformed_df_season_spring)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

transformed_df_season_summer = stats.boxcox(df_season_summer.sample(2500))[0]
test_stat, p_value = stats.shapiro(transformed_df_season_summer)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')

transformed_df_season_fall = stats.boxcox(df_season_fall.sample(2500))[0]
test_stat, p_value = stats.shapiro(transformed_df_season_fall)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')
```

```
transformed_df_season_winter = stats.boxcox(df_season_winter.sample(2500))[0]
test_stat, p_value = stats.shapiro(transformed_df_season_winter)
print('p-value:', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows normal distribution')
```

```
p-value: 0.0
The sample does not follow normal distribution
p-value: 1.0480194626653075e-37
The sample does not follow normal distribution
p-value: 1.5080510584110837e-35
The sample does not follow normal distribution
p-value: 2.657296010624101e-38
The sample does not follow normal distribution
p-value: 9.578545029083167e-17
The sample does not follow normal distribution
p-value: 4.155938766606108e-21
The sample does not follow normal distribution
p-value: 2.6401107051086862e-21
The sample does not follow normal distribution
p-value: 5.817313740136531e-20
The sample does not follow normal distribution
```

- Even after applying the Box-Cox transformation to each of the seasonal datasets, the samples still do not exhibit a normal distribution.

Homogeneity of Variances using Levene's test

In [59]:
```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = stats.levene(df_season_spring.sample(2500),
                                  df_season_summer.sample(2500),
                                  df_season_fall.sample(2500),
                                  df_season_winter.sample(2500))
print('p-value', p_value)
if p_value < 0.05:
    print('The samples do not have  Homogenous Variance')
else:
    print('The samples have Homogenous Variance ')
```

```
p-value 8.445709382946375e-105
The samples do not have  Homogenous Variance
```

Since the samples are not normally distributed and do not have the same variance, f_oneway test cannot be performed here, we can perform its non parametric equivalent test i.e., Kruskal-Wallis H-test for independent samples.

In [60]:
```
# Ho : Mean no. of cycles rented is same for different weather
# Ha : Mean no. of cycles rented is different for different weather
# Assuming significance Level to be 0.05
alpha = 0.05
test_stat, p_value = stats.kruskal(df_season_spring, df_season_summer, df_season_fall,
```

```
print('Test Statistic =', test_stat)
print('p value =', p_value)
```

```
Test Statistic = 699.6668548181988
p value = 2.479008372608633e-151
```

Comparing p value with significance level

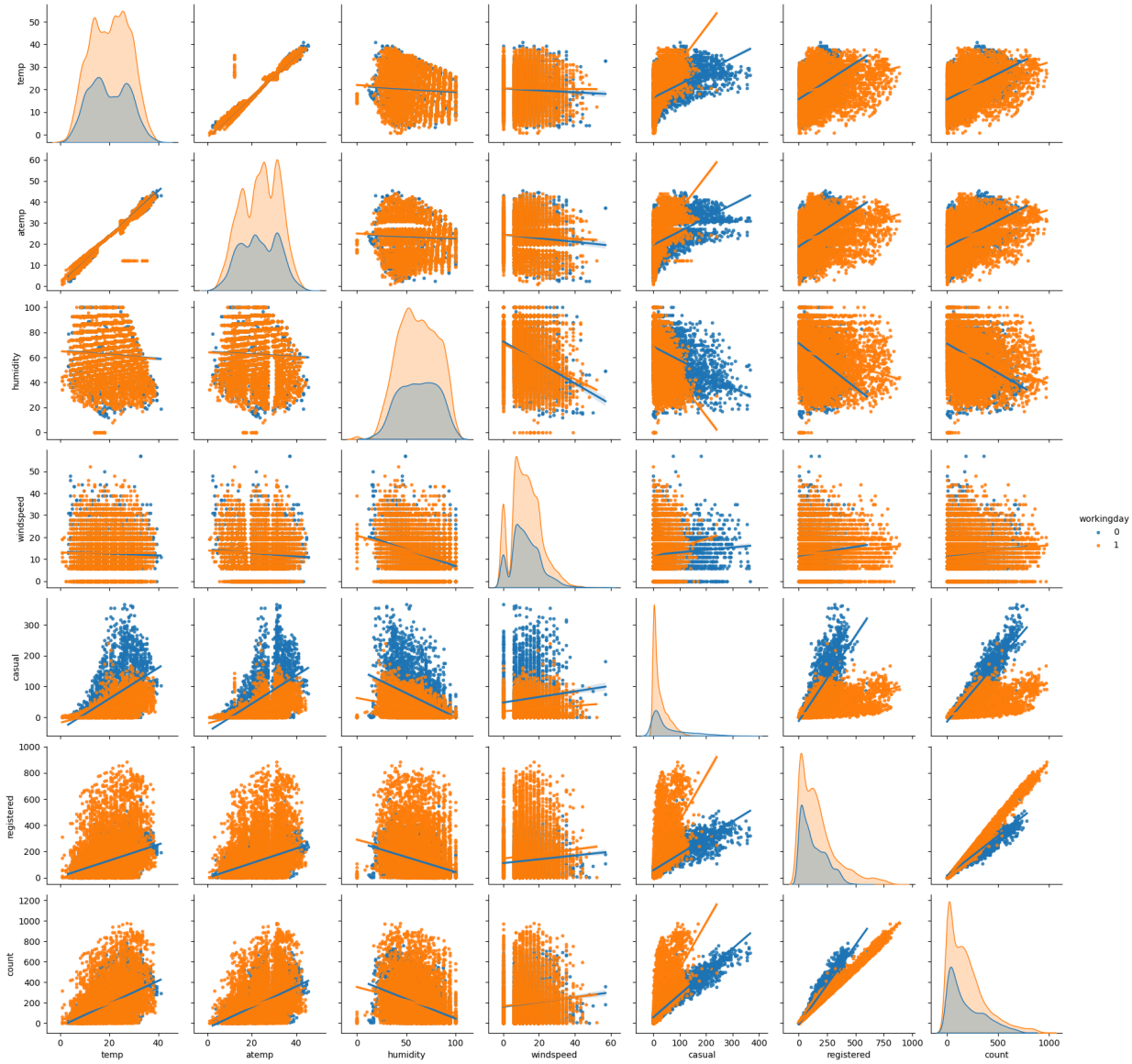
In [61]:
```
if p_value < alpha:
    print('Reject Null Hypothesis')
else:
    print('Failed to reject Null Hypothesis')
```

```
Reject Null Hypothesis
```

Therefore, the average number of rental bikes is statistically different for different seasons.

In [62]:
```
sns.pairplot(data = df,
             kind = 'reg',
             hue = 'workingday',
             markers = '.')
plt.plot()
```

Out[62]:
```
[]
```



In [63]:
```
corr_data = df.corr()
corr_data
```

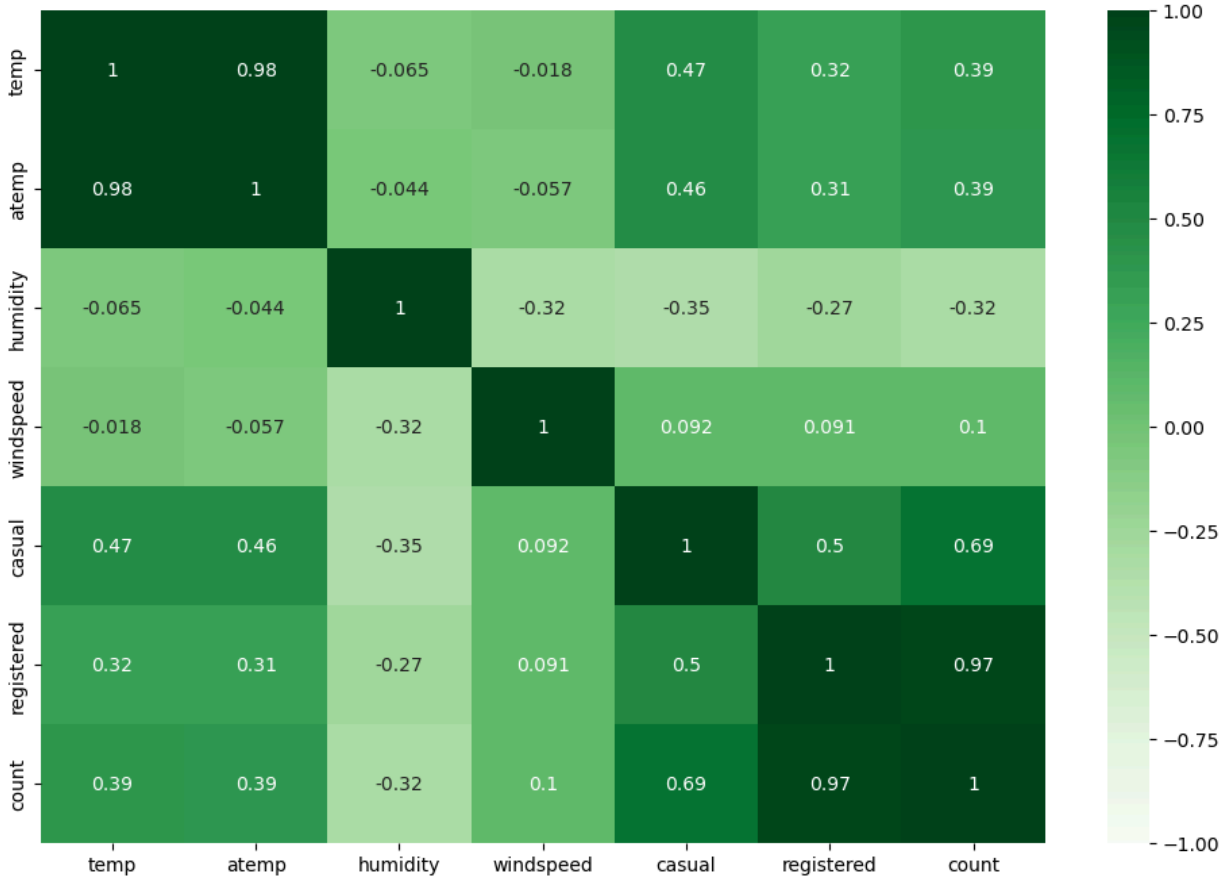
```
C:\Users\aa\AppData\Local\Temp\ipykernel_18200\919268980.py:1: FutureWarning: The def
ault value of numeric_only in DataFrame.corr is deprecated. In a future version, it w
ill default to False. Select only valid columns or specify the value of numeric_only
to silence this warning.
  corr_data = df.corr()
```

Out[63]:

	temp	atemp	humidity	windspeed	casual	registered	count
temp	1.000000	0.984948	-0.064949	-0.017852	0.467097	0.318571	0.394454
atemp	0.984948	1.000000	-0.043536	-0.057473	0.462067	0.314635	0.389784
humidity	-0.064949	-0.043536	1.000000	-0.318607	-0.348187	-0.265458	-0.317371
windspeed	-0.017852	-0.057473	-0.318607	1.000000	0.092276	0.091052	0.101369
casual	0.467097	0.462067	-0.348187	0.092276	1.000000	0.497250	0.690414
registered	0.318571	0.314635	-0.265458	0.091052	0.497250	1.000000	0.970948
count	0.394454	0.389784	-0.317371	0.101369	0.690414	0.970948	1.000000

In [64]:
```
plt.figure(figsize = (12, 8))
sns.heatmap(data = corr_data, cmap = 'Greens', annot = True, vmin = -1, vmax = 1)
plt.plot()
```

Out[64]:
```
[]
```



1. There is a very high correlation (> 0.9) between the columns [atemp, temp] and [count, registered].
2. There are no high positive or negative correlations (0.7 - 0.9) between any columns.
3. Moderate positive correlations (0.5 - 0.7) exist between the columns [casual, count] and [casual, registered].
4. Low positive correlations (0.3 - 0.5) exist between the columns [count, temp], [count, atemp], and [casual, atemp].
5. Negligible correlation exists between all other combinations of columns.

Inference from the analysis

1. The dataset spans from January 1, 2011, to December 19, 2012, covering a total duration of 718 days and 23 hours.

2. Out of every 100 users, approximately 19 are casual users and 81 are registered users.
3. The mean hourly count of rental bikes is 144 for the year 2011 and 239 for the year 2012, indicating an annual growth rate of 65.41%.
4. A seasonal pattern is observed in the count of rental bikes, with higher demand during the spring and summer months, followed by a slight decline in the fall and further decrease in the winter.
5. The average hourly count of rental bikes is lowest in January, followed by February and March.
6. Fluctuations in count occur throughout the day, with low counts during early morning hours, a sudden increase in the morning, a peak count in the afternoon, and a gradual decline in the evening and nighttime.
7. More than 80% of the time, the temperature is below 28 degrees Celsius.
8. Humidity levels are predominantly above 40%, indicating variations from optimum to too moist conditions for most of the time.
9. Windspeed data shows that over 85% of the total records have values less than 20.
10. The highest hourly count of total rental bikes is observed during clear and cloudy weather, followed by misty weather and rainy weather, with few records for extreme weather conditions.
11. The mean hourly count of total rental bikes is statistically similar for both working and non-working days.
12. There is a statistically significant dependency between weather and season based on the hourly total number of bikes rented.
13. The hourly total number of rental bikes varies significantly across different weather conditions.
14. There is no statistically significant dependency of weather types 1, 2, and 3 on season based on the average hourly total number of bikes rented.
15. The hourly total number of rental bikes differs significantly across different seasons.

Recommendations

1. **Seasonal Marketing:** Adjust marketing strategies to promote bike rentals during spring and summer months when demand is high. Offer seasonal discounts to attract more customers during these periods.

2. **Time-based Pricing:** Implement time-based pricing to encourage rentals during off-peak hours with lower rates and balance demand throughout the day.

3. **Weather-based Promotions:** Offer weather-specific discounts during clear and cloudy conditions, which see higher rental counts, to attract more customers.

4. **User Segmentation:** Tailor marketing strategies for registered and casual users. Provide loyalty programs for registered users and focus on seamless rental experiences for casual users.

5. **Optimize Inventory:** Analyze demand patterns and adjust inventory levels accordingly to avoid excess bikes during low-demand months and ensure availability during peak months.

6. **Improve Weather Data Collection:** Enhance data collection for extreme weather conditions to better understand customer behavior and adjust operations accordingly.

7. **Customer Comfort:** Provide amenities like umbrellas, rain jackets, and water bottles to enhance customer comfort and convenience.

8. **Collaborations with Weather Services:** Partner with weather services to provide real-time updates and forecasts to potential customers, incorporating weather information into marketing campaigns.

9. **Seasonal Bike Maintenance:** Allocate resources for seasonal bike maintenance to ensure bikes are in top condition before peak seasons and conduct regular inspections throughout the year.

10. **Customer Feedback and Reviews:** Encourage customers to provide feedback to identify areas for improvement and tailor services to meet customer expectations.

11. **Social Media Marketing:** Utilize social media platforms to promote electric bike rental services with engaging content, customer testimonials, and targeted advertising campaigns.

12. **Special Occasion Discounts:** Offer special discounts on occasions like Zero Emissions Day, Earth Day, and World Environment Day to attract new users and promote sustainability.