Business Case: Yulu - Hypothesis Testing

About Yulu

Yulu is India's leading ***micro-mobility service provider***, which ***offers unique vehicles for the daily commute***. Starting off as a ***mission to eliminate traffic congestion in India***, Yulu provides the safest commute solution through a ***user-friendly mobile app to enable shared, solo and sustainable commuting.***

Yulu zones are located at all the appropriate locations (including ***metro stations, bus stands, office spaces, residential areas, corporate offices, etc***) to make those first and last miles smooth, affordable, and convenient!

Yulu has recently suffered considerable dips in its revenues. They have contracted a consulting company to understand the factors on which the demand for these shared electric cycles depends. Specifically, they want ***to understand the factors affecting the demand for these shared electric cycles 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

```
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 spy
```

Reading the dataset

```
df = pd.read_csv("bike_sharing.csv")
df
```

	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	0.0000	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0000	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0000	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0000	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0000	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...
10881	2012-12-19 19:00:00	4	0	1	1	15.58	19.695	50	26.0027	7	329	336
10882	2012-12-19 20:00:00	4	0	1	1	14.76	17.425	57	15.0013	10	231	241
10883	2012-12-19 21:00:00	4	0	1	1	13.94	15.910	61	15.0013	4	164	168
10884	2012-12-19 22:00:00	4	0	1	1	13.94	17.425	61	6.0032	12	117	129
10885	2012-12-19 23:00:00	4	0	1	1	13.12	16.665	66	8.9981	4	84	88

10886 rows × 12 columns

Next steps: Generate code with df View recommended plots

Shape of the dataset

```
df.shape
```

```
(10886, 12)
```

Columns in the Dataset

```
df.columns
```

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

Basic information about the values present in the dataset

```
df.head(10)
```

	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	0.0000	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0000	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0000	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0000	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0000	0	1	1
5	2011-01-01 05:00:00	1	0	0	2	9.84	12.880	75	6.0032	0	1	1
6	2011-01-01 06:00:00	1	0	0	1	9.02	13.635	80	0.0000	2	0	2
7	2011-01-01 07:00:00	1	0	0	1	8.20	12.880	86	0.0000	1	2	3
8	2011-01-01 08:00:00	1	0	0	1	9.84	14.395	75	0.0000	1	7	8
9	2011-01-01 09:00:00	1	0	0	1	13.12	17.425	76	0.0000	8	6	14

Next steps: Generate code with df | View recommended plots

```
df.tail(10)
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
10876	2012-12-19 14:00:00	4	0	1	1	17.22	21.210	50	12.9980	33	185	218
10877	2012-12-19 15:00:00	4	0	1	1	17.22	21.210	50	19.0012	28	209	237
10878	2012-12-19 16:00:00	4	0	1	1	17.22	21.210	50	23.9994	37	297	334
10879	2012-12-19 17:00:00	4	0	1	1	16.40	20.455	50	26.0027	26	536	562
10880	2012-12-19 18:00:00	4	0	1	1	15.58	19.695	50	23.9994	23	546	569
10881	2012-12-19 19:00:00	4	0	1	1	15.58	19.695	50	26.0027	7	329	336
10882	2012-12-19 20:00:00	4	0	1	1	14.76	17.425	57	15.0013	10	231	241
10883	2012-12-19 21:00:00	4	0	1	1	13.94	15.910	61	15.0013	4	164	168
10884	2012-12-19 22:00:00	4	0	1	1	13.94	17.425	61	6.0032	12	117	129
10885	2012-12-19 23:00:00	4	0	1	1	13.12	16.665	66	8.9981	4	84	88

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

Is there any null value in the dataset ?

```
np.any(df.isna())
```

```
False
```

Is there any duplicated values in the dataset ?

```
np.any(df.duplicated())
```

```
False
```

Datatype of the columns

```
df.dtypes
```

```
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
```

Converting the datatype of datetime column from object to datetime

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

What is the time period for which the data is given ?

```
df['datetime'].min()
```

```
Timestamp('2011-01-01 00:00:00')
```

```
df['datetime'].max()
```

```
Timestamp('2012-12-19 23:00:00')
```

```
df['datetime'].max() - df['datetime'].min()
```

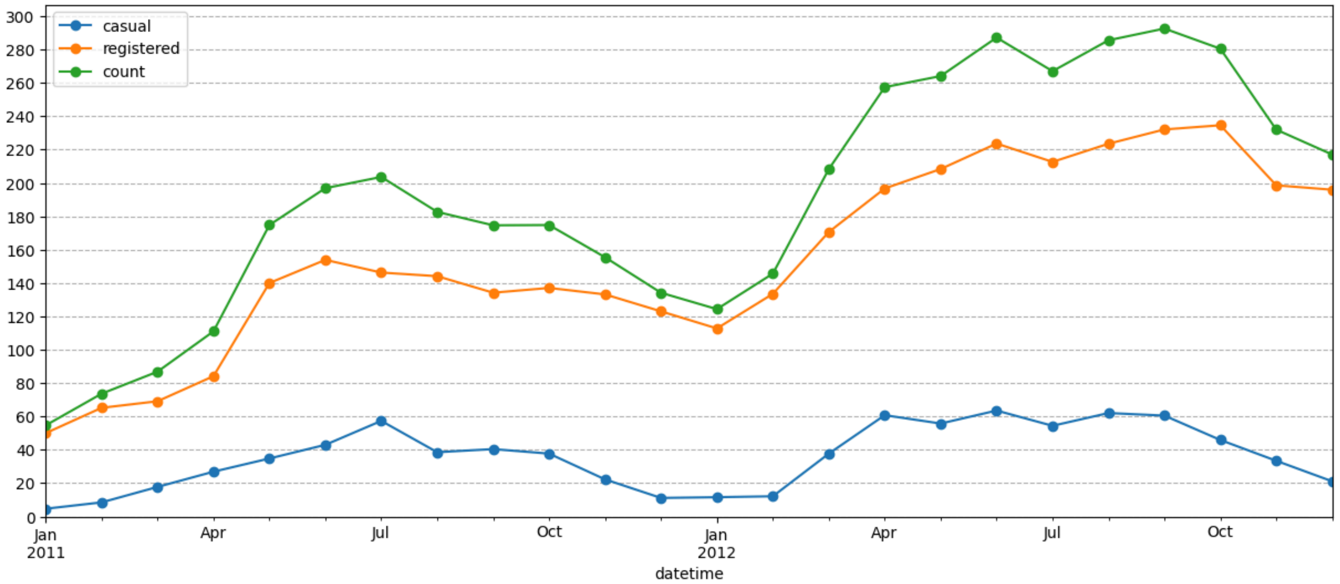
```
Timedelta('718 days 23:00:00')
```

```
df['day'] = df['datetime'].dt.day_name()
```

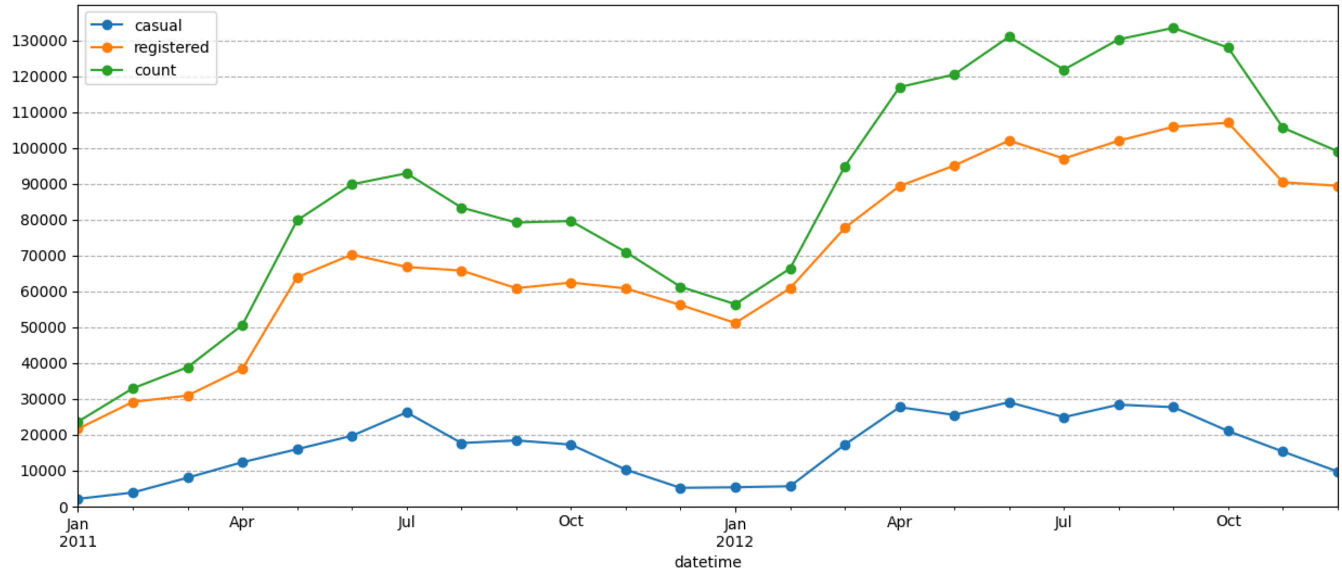
```
# setting the 'datetime' column as the index of the DataFrame 'df'
df.set_index('datetime', inplace = True)

# 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.
```

Slicing Data by Time

```
# The below code visualizes the trend of the monthly average values for the 'casual', 'registered',
    # and 'count' variables,  allowing for easy comparison and analysis of their patterns over time

plt.figure(figsize = (15, 6))

# plotting a lineplot by resampling the data on a monthly basis, and calculating the mean value
    # of 'casual', 'registered' and 'count' users for each month
df.resample('M')['casual'].mean().plot(kind = 'line', legend = 'casual', marker = 'o')
df.resample('M')['registered'].mean().plot(kind = 'line', legend = 'registered', marker = 'o')
df.resample('M')['count'].mean().plot(kind = 'line', legend = 'count', marker = 'o')

plt.grid(axis = 'y', linestyle = '--')    # adding gridlines only along the y-axis
plt.yticks(np.arange(0, 301, 20))
plt.ylim(0,)    # setting the lower y-axis limit to 0
plt.show()      # displaying the plot
```




```
# The below code visualizes the trend of the monthly total values for the 'casual', 'registered',
    # and 'count' variables,  allowing for easy comparison and analysis of their patterns over time

plt.figure(figsize = (15, 6))

# plotting a lineplot by resampling the data on a monthly basis, and calculating the sum
    # of 'casual', 'registered' and 'count' users for each month
df.resample('M')['casual'].sum().plot(kind = 'line', legend = 'casual', marker = 'o')
df.resample('M')['registered'].sum().plot(kind = 'line', legend = 'registered', marker = 'o')
df.resample('M')['count'].sum().plot(kind = 'line', legend = 'count', marker = 'o')

plt.grid(axis = 'y', linestyle = '--')      # adding gridlines only along the y-axis
plt.yticks(np.arange(0, 130001, 10000))
plt.ylim(0,)        # setting the lower y-axis limit to 0
plt.show()          # displaying the plot
```



I want to know if there is an increase in the average hourly count of rental bikes from the year 2011 to 2012

```
# 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 year
df1['growth_percent'] = (df1['count'] - df1['prev_count']) * 100 / df1['prev_count']
df1
```

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

Next steps: Generate code with df1 View recommended plots

- This data suggests that there was substantial growth in the count of the variable over the course of one year.
- The mean total hourly count of rental bikes is 144 for the year 2011 and 239 for the year 2012. An annual growth rate of 65.41 % can be seen in the demand of electric vehicles on an hourly basis.

It indicates positive growth and potentially a successful outcome or increasing demand for the variable being measured.

```
df.reset_index(inplace = True)
```

How does the average hourly count of rental bikes varies for different month ?

```
# Grouping the DataFrame by the month
df1 = df.groupby(by = df['datetime'].dt.month)['count'].mean().reset_index()
df1.rename(columns = {'datetime' : 'month'}, inplace = True)

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

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

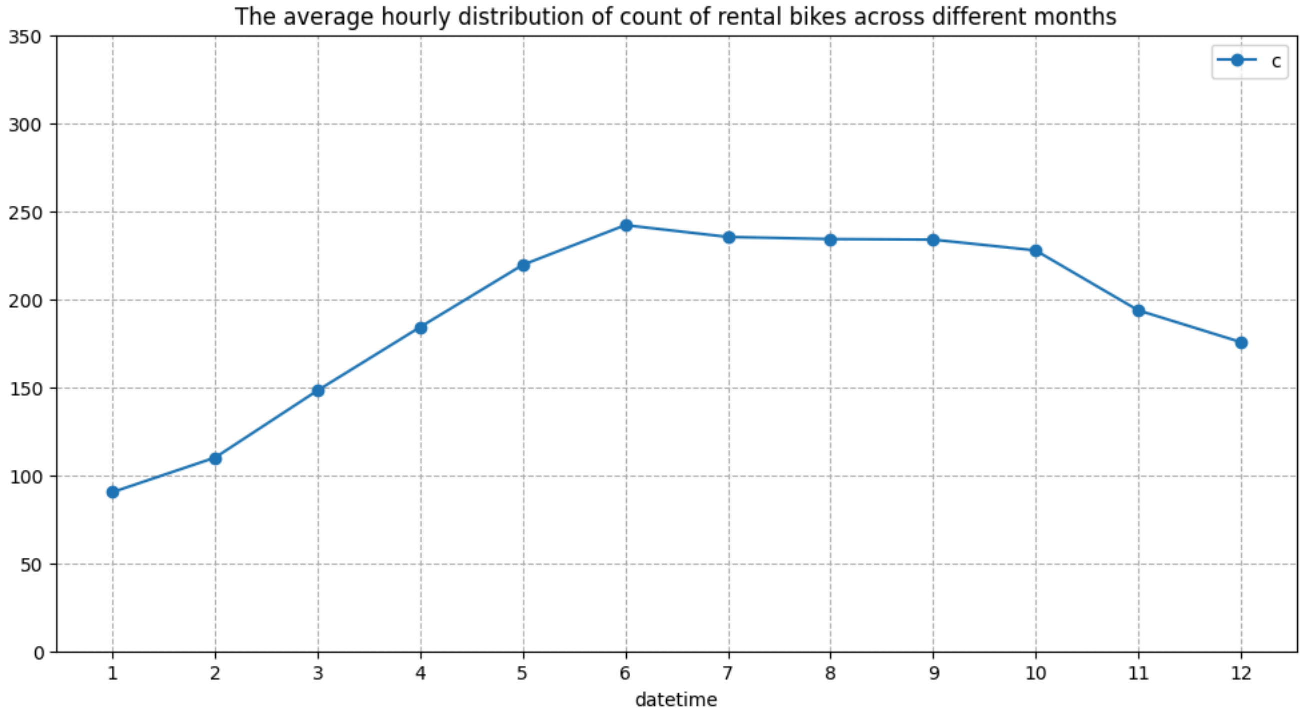
month	count	prev_count	growth_percent
1	90.366516	NaN	NaN
2	110.003330	90.366516	21.730188
3	148.169811	110.003330	34.695751
4	184.160616	148.169811	24.290241
5	219.459430	184.160616	19.167406
6	242.031798	219.459430	10.285440
7	235.325658	242.031798	-2.770768
8	234.118421	235.325658	-0.513007
9	233.805281	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

Next steps: Generate code with df1 View recommended plots

- The count of rental bikes shows an increasing trend from January to March, with a significant growth rate of 34.70% between February and March.
- The growth rate starts to stabilize from April to June, with a relatively smaller growth rate.
- From July to September, there is a slight decrease in the count of rental bikes, with negative growth rates.
- The count further declines from October to December, with the largest drop observed between October and November (-14.94%).

```
# The resulting plot visualizes the average hourly distribution of the count of rental bikes for each
    # month, allowing for comparison and identification of any patterns or trends throughout the year.

# 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 months")

# Grouping the DataFrame by the month and calculating the mean of the 'count' column for each month.
    # Ploting the line graph using markers ('o') to represent the average count per month.
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 12
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.
```

[]



- The average hourly count of rental bikes is the highest in the month of June followed by July and August.
- The average hourly count of rental bikes is the lowest in the month of January followed by February and March.

Overall, these trends suggest a seasonal pattern in the count of rental bikes, with higher demand during the spring and summer months, a slight decline in the fall, and a further decrease in the winter months. It could be useful for the rental bike company to consider these patterns for resource allocation, marketing strategies, and operational planning throughout the year.

What is the distribution of average count of rental bikes on an hourly basis in a single day ?

```
# Grouping the DataFrame by the hour
df1 = df.groupby(by = df['datetime'].dt.hour)['count'].mean().reset_index()
df1.rename(columns = {'datetime' : 'hour'}, inplace = True)

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

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

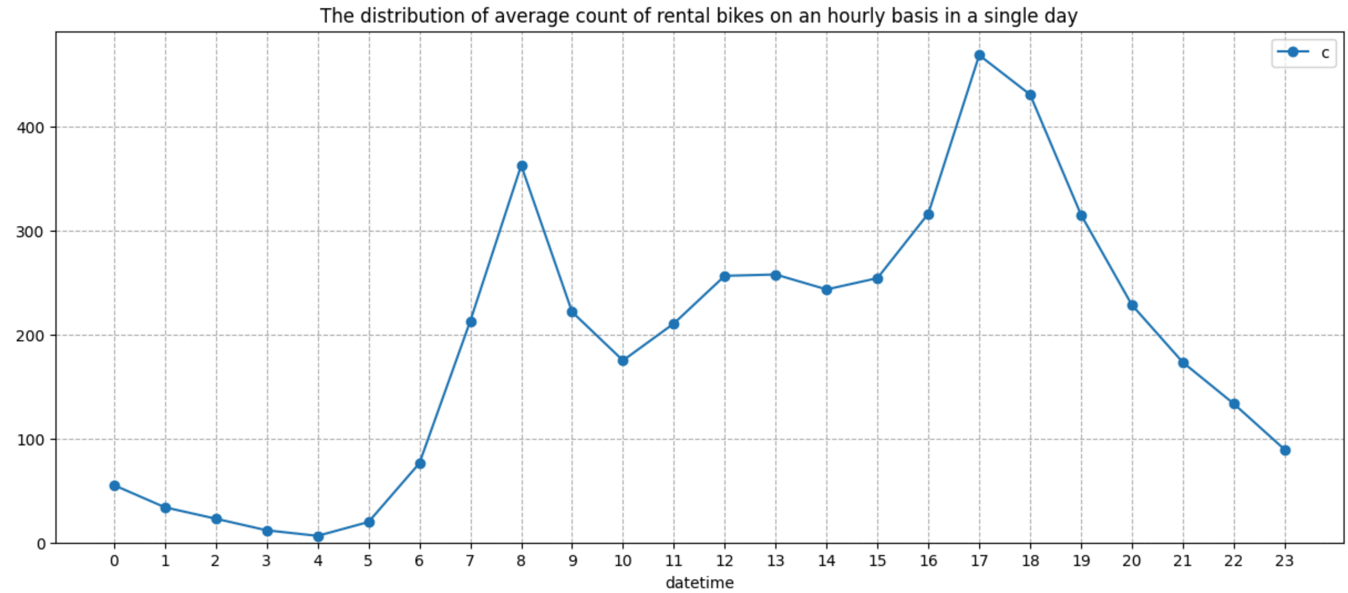
hour	count	prev_count	growth_percent
0	55.138462	NaN	NaN
1	33.859031	55.138462	-38.592718
2	22.899554	33.859031	-32.367959
3	11.757506	22.899554	-48.656179
4	6.407240	11.757506	-45.505110
5	19.767699	6.407240	208.521293
6	76.259341	19.767699	285.777526
7	213.116484	76.259341	179.462793
8	362.769231	213.116484	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

Next steps: Generate code with df1 View recommended plots

- 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.

```
plt.figure(figsize = (15, 6))
plt.title("The distribution of average count of rental bikes on an hourly basis in a single day")
df.groupby(by = df['datetime'].dt.hour)['count'].mean().plot(kind = 'line', marker = 'o')
plt.ylim(0,)
plt.xticks(np.arange(0, 24))
plt.legend('count')
plt.grid(axis = 'both', linestyle = '--')
plt.plot()
```

[]



- The average count of rental bikes is the highest at 5 PM followed by 6 PM and 8 AM of the day.
- The average count of rental bikes is the lowest at 4 AM followed by 3 AM and 5 AM of the day.

These patterns indicate that there is a distinct fluctuation in count 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.

Basic Information about the Dataset

```
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  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: category(4), datetime64[ns](1), float32(1), float64(2), int64(4), object(1)
memory usage: 766.2+ KB
```

- The dataframe requires a memory usage of about 1.1+ MB.
- Though the memory usage is small but can we still decrease the memory usage ?

```
# 1: spring, 2: summer, 3: fall, 4: winter
def season_category(x):
    if x == 1:
        return 'spring'
    elif x == 2:
        return 'summer'
    elif x == 3:
        return 'fall'
    else:
        return 'winter'
df['season'] = df['season'].apply(season_category)
```

Optimizing Memory Usage of the Dataframe

Updating dtype of season column

```
print('Memory usage of season column : ', df['season'].memory_usage())
# Since the dtype of season column is object, we can convert the dtype to category to save memory
df['season'] = df['season'].astype('category')
print('Updated Memory usage of season column : ', df['season'].memory_usage())
```

```
Memory usage of season column :  87216
Updated Memory usage of season column :  11218
```

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())
# Since the maximum entry in holiday column is 1 and the dtype is int64, we can convert the dtype to category to save memory
df['holiday'] = df['holiday'].astype('category')
print('Updated Memory usage of holiday column : ', df['holiday'].memory_usage())
```

```
Max value entry in holiday column :  1
Memory usage of holiday column :  87216
Updated Memory usage of holiday column :  11138
```

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())
# Since the maximum entry in workingday column is 1 and the dtype is int64, we can convert the dtype to category to save memory
df['workingday'] = df['workingday'].astype('category')
print('Updated Memory usage of workingday column : ', df['workingday'].memory_usage())
```

```
Max value entry in workingday column :  1
Memory usage of workingday column :  87216
Updated Memory usage of workingday column :  11138
```

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())
# Since the maximum entry in weather column is 4 and the dtype is int64, we can convert the dtype to category to save memory
df['weather'] = df['weather'].astype('category')
print('Updated Memory usage of weather column : ', df['weather'].memory_usage())
```

```
Max value entry in weather column :  4
Memory usage of weather column :  87216
Updated Memory usage of weather column :  11218
```

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())
# Since the maximum entry in temp column is 41.0 and the dtype is float64, we can convert the dtype to float32 to save memory
df['temp'] = df['temp'].astype('float32')
print('Updated Memory usage of temp column : ', df['temp'].memory_usage())
```

```
Max value entry in temp column :  41.0
Memory usage of temp column :  87216
Updated Memory usage of temp column :  43672
```

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())
# Since the maximum entry in atemp column is 45.455 and the dtype is float64, we can convert the dtype to float32 to save memory
df['atemp'] = df['atemp'].astype('float32')
print('Updated Memory usage of atemp column : ', df['atemp'].memory_usage())
```

```
Max value entry in atemp column :  45.455
Memory usage of atemp column :  87216
```

```
Updated Memory usage of atemp column :  43672
```

Updating dtype of humidity column

```
print('Max value entry in humidity column : ', df['humidity'].max())
print('Memory usage of humidity column : ', df['temp'].memory_usage())
# Since the maximum entry in humidity column is 100 and the dtype is int64, we can convert the dtype to int8 to save memory
df['humidity'] = df['humidity'].astype('int8')
print('Updated Memory usage of humidity column : ', df['humidity'].memory_usage())
```

```
Max value entry in humidity column :  100
Memory usage of humidity column :  43672
Updated Memory usage of humidity column :  11014
```

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())
# Since the maximum entry in windspeed column is 56.9969 and the dtype is float64, we can convert the dtype to float32 to save memory
df['windspeed'] = df['windspeed'].astype('float32')
print('Updated Memory usage of windspeed column : ', df['windspeed'].memory_usage())
```

```
Max value entry in windspeed column :  56.9969
Memory usage of windspeed column :  87216
Updated Memory usage of windspeed column :  43672
```

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())
# Since the maximum entry in casual column is 367 and the dtype is int64, we can convert the dtype to int16 to save memory
df['casual'] = df['casual'].astype('int16')
print('Updated Memory usage of casual column : ', df['casual'].memory_usage())
```

```
Max value entry in casual column :  367
Memory usage of casual column :  87216
Updated Memory usage of casual column :  21900
```

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())
# Since the maximum entry in registered column is 886 and the dtype is int64, we can convert the dtype to int16 to save memory
df['registered'] = df['registered'].astype('int16')
print('Updated Memory usage of registered column : ', df['registered'].memory_usage())
```

```
Max value entry in registered column :  886
Memory usage of registered column :  87216
Updated Memory usage of registered column :  21900
```

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())
# Since the maximum entry in count column is 977 and the dtype is int64, we can convert the dtype to int16 to save memory
df['count'] = df['count'].astype('int16')
print('Updated Memory usage of count column : ', df['count'].memory_usage())
```

```
Max value entry in count column :  977
Memory usage of count column :  87216
Updated Memory usage of count column :  21900
```

```
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  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
dtypes: category(4), float32(3), int16(3), int8(1), object(1)
memory usage: 330.3+ KB
```

Earlier the dataset was using 1.1+ MB of memory but now it has been reduced to 415.2+ KB. Around 63.17 % reduction in the memory usage.

Basic Description of the dataset

```
df.describe()
```

	temp	atemp	humidity	windspeed	casual	registered	count
count	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000
mean	20.230862	23.655085	61.886460	12.799396	36.021955	155.552177	191.574132
std	7.791590	8.474601	19.245033	8.164537	49.960477	151.039033	181.144454
min	0.820000	0.760000	0.000000	0.000000	0.000000	0.000000	1.000000
25%	13.940000	16.665001	47.000000	7.001500	4.000000	36.000000	42.000000
50%	20.500000	24.240000	62.000000	12.998000	17.000000	118.000000	145.000000
75%	26.240000	31.059999	77.000000	16.997900	49.000000	222.000000	284.000000
max	41.000000	45.455002	100.000000	56.996899	367.000000	886.000000	977.000000

- These statistics provide insights into the central tendency, spread, and range of the numerical features in the dataset.

```
np.round(df['season'].value_counts(normalize = True) * 100, 2)
```

```
season
winter    25.11
fall      25.11
summer    25.11
spring    24.67
Name: proportion, dtype: float64
```

```
np.round(df['holiday'].value_counts(normalize = True) * 100, 2)
```

```
holiday
0    97.14
1     2.86
Name: proportion, dtype: float64
```

```
np.round(df['workingday'].value_counts(normalize = True) * 100, 2)
```

```
workingday
1    68.09
0    31.91
Name: proportion, dtype: float64
```

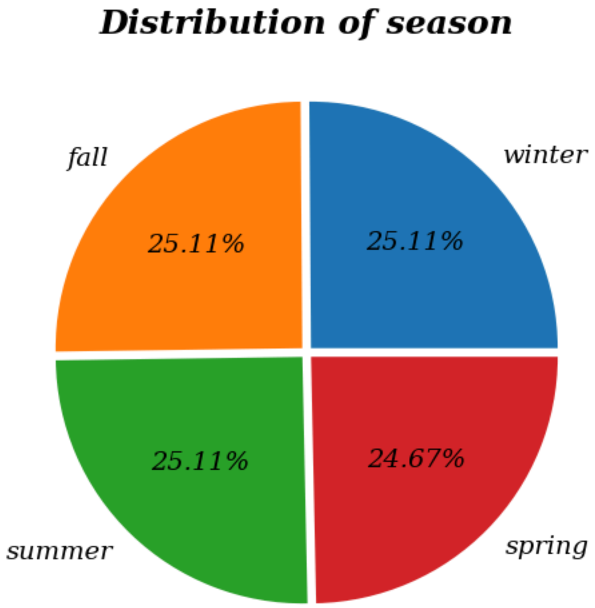
```
np.round(df['weather'].value_counts(normalize = True) * 100, 2)
```

```
weather
1    66.07
2    26.03
3     7.89
4     0.01
Name: proportion, dtype: float64
```

```
# The below code generates a visually appealing pie chart to showcase the
    # distribution of seasons in the dataset

plt.figure(figsize = (6, 6))      # setting the figure size to 6*6

# setting the title of the plot
plt.title('Distribution of season', fontdict = {'fontsize' : 18,
                                                'fontweight' : 600,
                                                'fontstyle' : 'oblique',
                                                'fontfamily' : 'serif'})

df_season = np.round(df['season'].value_counts(normalize = True) * 100, 2).to_frame()
```

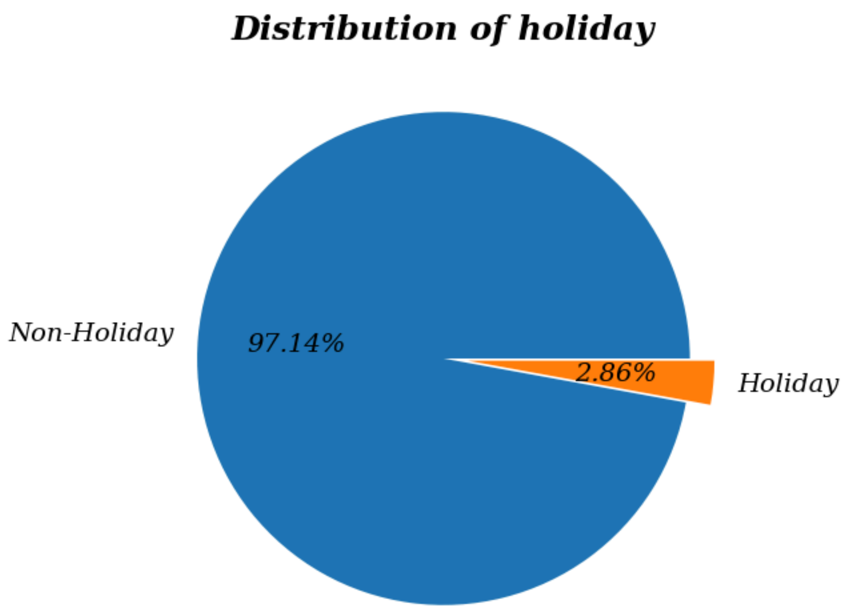
```
df_season
# Creating the pie-chart
plt.pie(x = df_season['proportion'],
        explode = [0.025, 0.025, 0.025, 0.025],
        labels = df_season.index,
        autopct = '%.2f%%',
        textprops = {'fontsize' : 14,
                   'fontstyle' : 'oblique',
                   'fontfamily' : 'serif',
                   'fontweight' : 500})

plt.plot()    # displaying the plot
```

[]



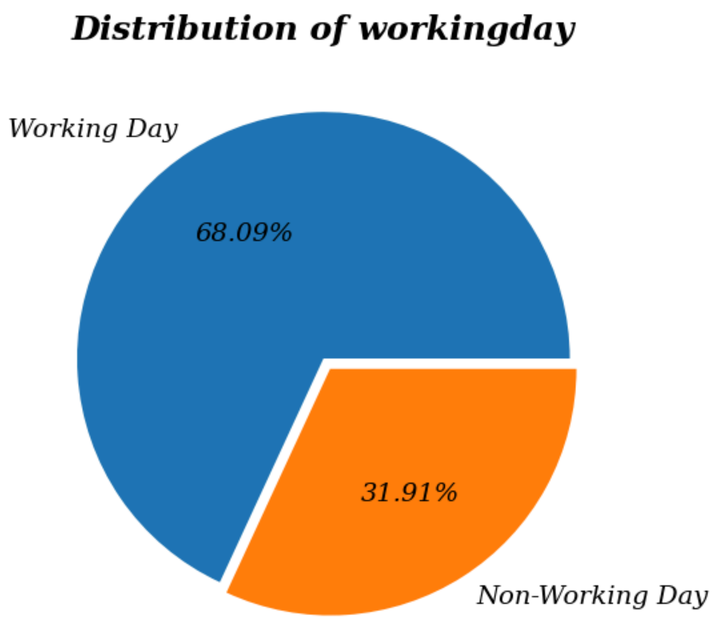

```
# The below code generates a visually appealing pie chart to showcase the
    # distribution of holiday in the dataset

plt.figure(figsize = (6, 6))     # setting the figure size to 6*6

# setting the title of the plot
plt.title('Distribution of holiday', fontdict = {'fontsize' : 18,
                                                 'fontweight' : 600,
                                                 'fontstyle' : 'oblique',
                                                 'fontfamily' : 'serif'})

df_holiday = np.round(df['holiday'].value_counts(normalize = True) * 100, 2).to_frame()

# Creating the pie-chart
plt.pie(x = df_holiday['proportion'],
        explode = [0, 0.1],
        labels = ['Non-Holiday', 'Holiday'],
        autopct = '%.2f%%',
        textprops = {'fontsize' : 14,
                   'fontstyle' : 'oblique',
                   'fontfamily' : 'serif',
                   'fontweight' : 500})

plt.plot()       # displaying the plot
```

[]



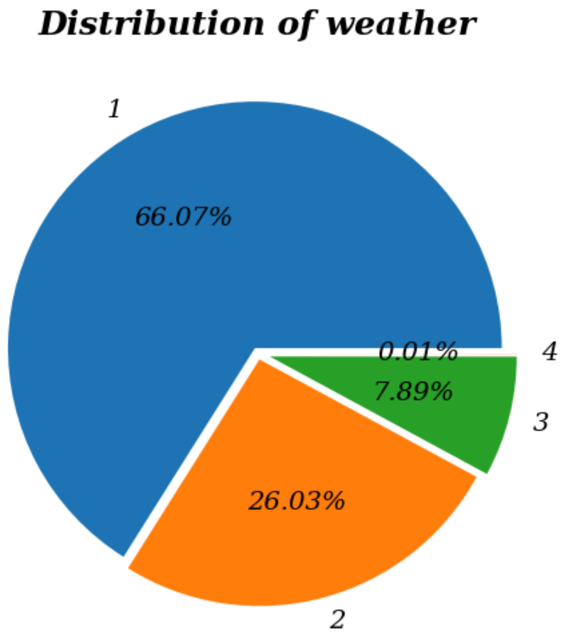

```
# The below code generates a visually appealing pie chart to showcase the
    # distribution of workingday in the dataset

plt.figure(figsize = (6, 6))    # setting the figure size to 6*6

# setting the title of the plot
plt.title('Distribution of workingday', fontdict = {'fontsize' : 18,
                                                    'fontweight' : 600,
                                                    'fontstyle' : 'oblique',
                                                    'fontfamily' : 'serif'})

df_workingday = np.round(df['workingday'].value_counts(normalize = True) * 100, 2).to_frame()

# Creating the pie-chart
plt.pie(x = df_workingday['proportion'],
        explode = [0, 0.05],
        labels = ['Working Day', 'Non-Working Day'],
        autopct = '%.2f%%',
        textprops = {'fontsize' : 14,
                   'fontstyle' : 'oblique',
                   'fontfamily' : 'serif',
                   'fontweight' : 500})

plt.plot()         # displaying the plot
```

[]




```
# The below code generates a visually appealing pie chart to showcase the
    # distribution of weather in the dataset

plt.figure(figsize = (6, 6))     # setting the figure size to 6*6

# setting the title of the plot
plt.title('Distribution of weather', fontdict = {'fontsize' : 18,
                                                 'fontweight' : 600,
                                                 'fontstyle' : 'oblique',
                                                 'fontfamily' : 'serif'})

df_weather = np.round(df['weather'].value_counts(normalize = True) * 100, 2).to_frame()

# Creating the pie-chart
plt.pie(x = df_weather['proportion'],
        explode = [0.025, 0.025, 0.05, 0.05],
        labels = df_weather.index,
        autopct = '%.2f%%',
        textprops = {'fontsize' : 14,
                     'fontstyle' : 'oblique',
                     'fontfamily' : 'serif',
                     'fontweight' : 500})

plt.plot()        # displaying the plot
```

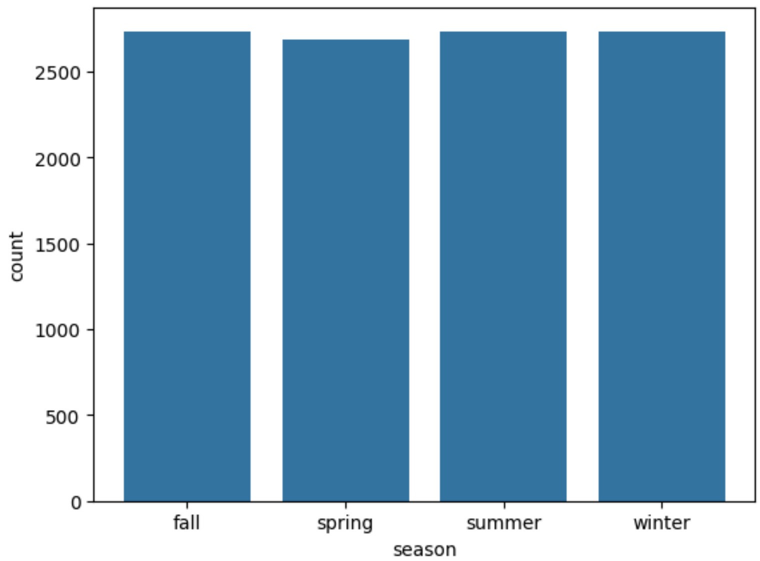
[]



Univariate Analysis

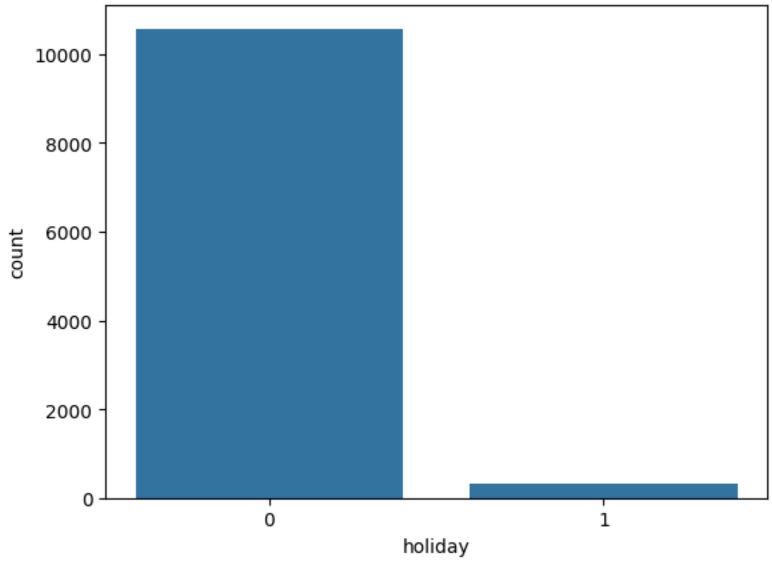
```
# The below code generates a visually appealing count plot to showcase the
    # distribution of season in the dataset
sns.countplot(data = df, x = 'season')
plt.plot()    # displaying the plot
```

[]



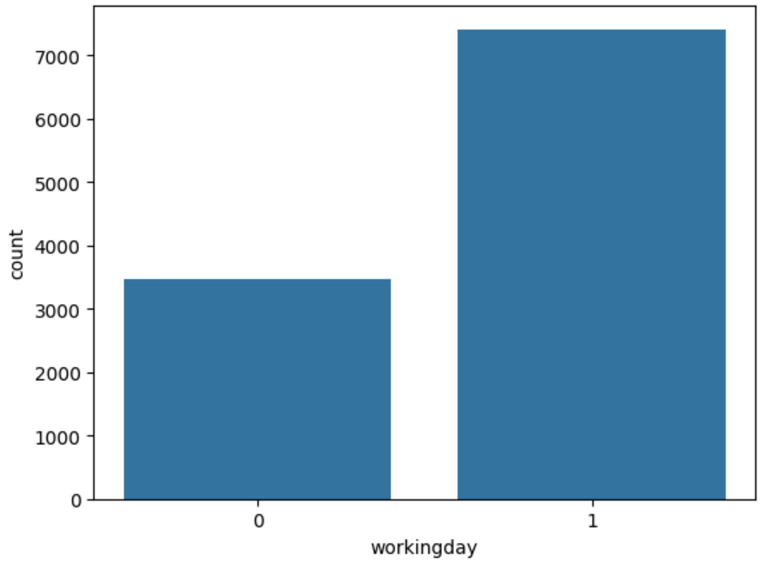

```
# The below code generates a visually appealing count plot to showcase the
    # distribution of holiday in the dataset

sns.countplot(data = df, x = 'holiday')
plt.plot()       # displaying the chart
```

[]



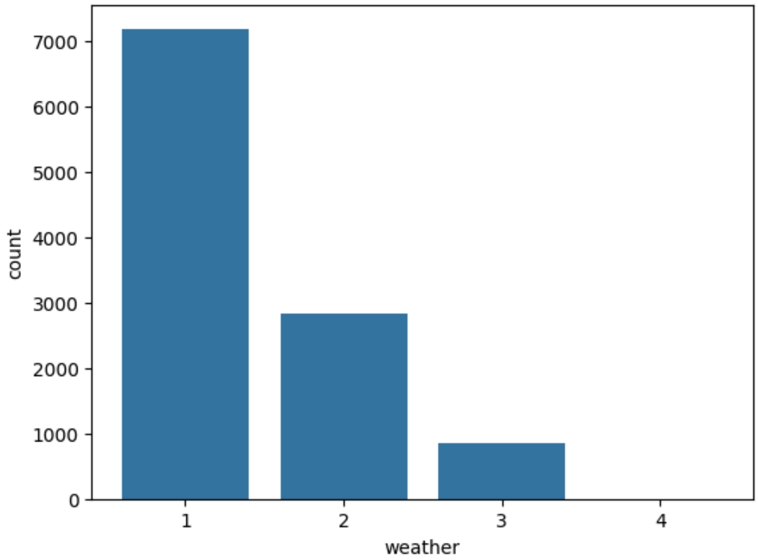

```
# The below code generates a visually appealing count plot to showcase the
    # distribution of workingday in the dataset

sns.countplot(data = df, x = 'workingday')
plt.plot()      # displaying the chart
```

[]



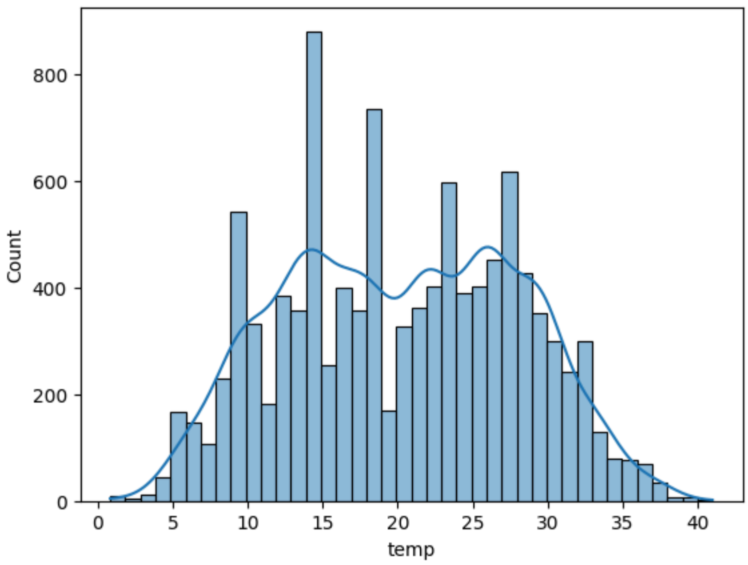

```
# The below code generates a visually appealing count plot to showcase the
    # distribution of weather in the dataset

sns.countplot(data = df, x = 'weather')
plt.plot()        # displaying the chart
```

```
[]
```




```
# The below code generates a histogram plot for the 'temp' feature, showing the distribution of
    # temperature values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'temp', kde = True, bins = 40)
plt.plot()        # displaying the chart
```

```
[]
```



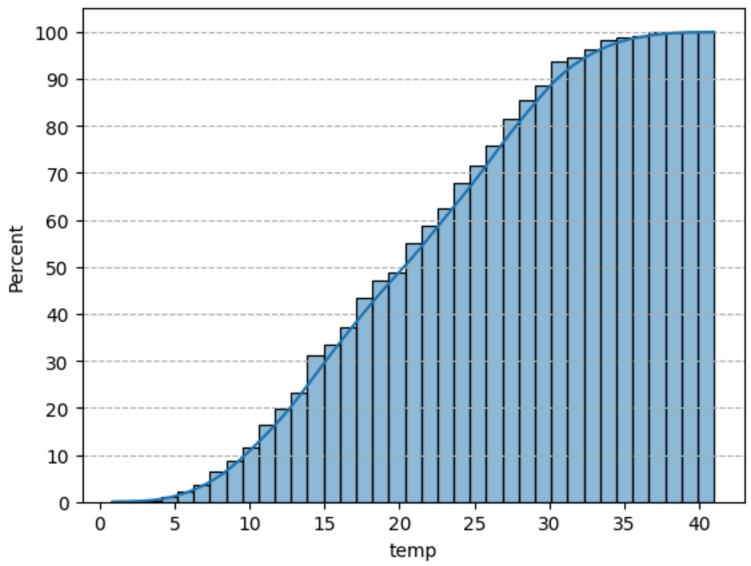

```
temp_mean = np.round(df['temp'].mean(), 2)
temp_std = np.round(df['temp'].std(), 2)
temp_mean, temp_std
```

```
(20.23, 7.79)
```

- The mean and the standard deviation of the temp column is 20.23 and 7.79 degree celcius respectively.

```
# The below code generates a histogram plot for the 'temp' feature, showing the cumulative
    # distribution of temperature values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'temp', kde = True, cumulative = True, stat = 'percent')
plt.grid(axis = 'y', linestyle = '--')
plt.yticks(np.arange(0, 101, 10))
plt.plot()          # displaying the chart
```

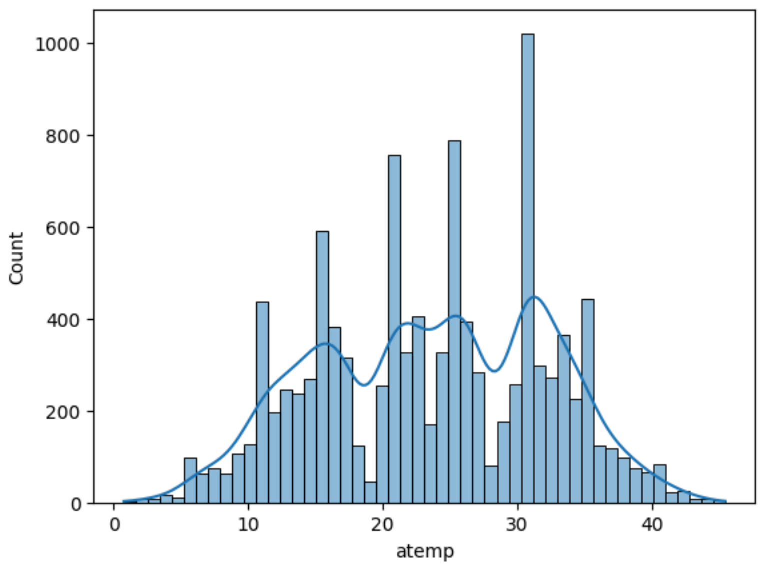
[]



- More than 80 % of the time, the temperature is less than 28 degrees celcius.

```
# The below code generates a histogram plot for the 'atemp' feature, showing the distribution of
    # feeling temperature values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'atemp', kde = True, bins = 50)
plt.plot()         # displaying the chart
```

[]



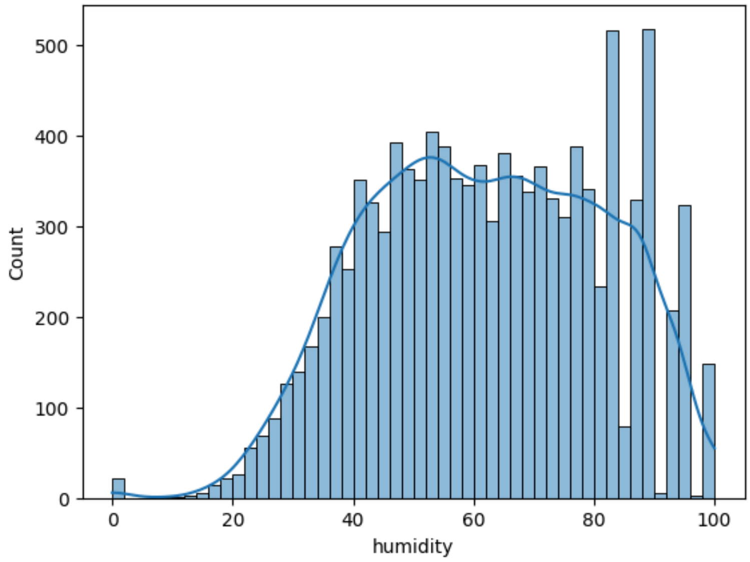

```
temp_mean = np.round(df['atemp'].mean(), 2)
temp_std = np.round(df['atemp'].std(), 2)
temp_mean, temp_std
```

(23.66, 8.47)

- The mean and the standard deviation of the atemp column is 23.66 and 8.47 degree celcius respectively.

```
# The below code generates a histogram plot for the 'humidity' feature, showing the distribution of
    # humidity values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'humidity', kde = True, bins = 50)
plt.plot()       # displaying the chart
```

[]



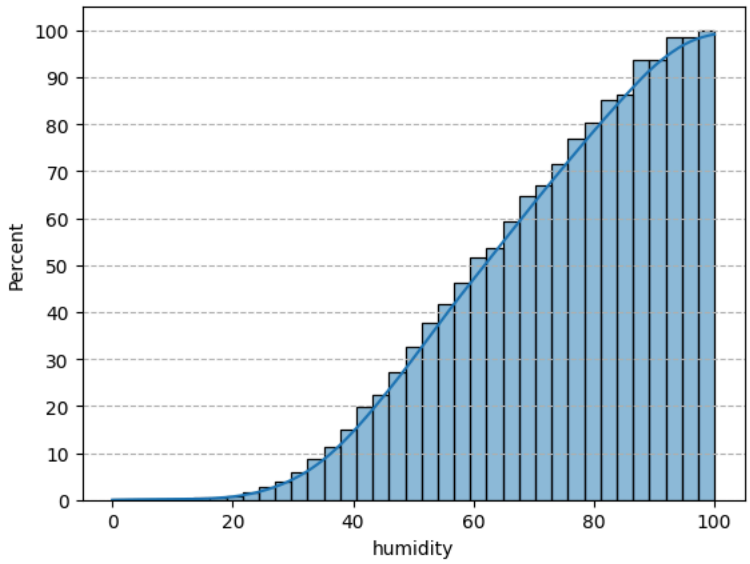

```
humidity_mean = np.round(df['humidity'].mean(), 2)
humidity_std = np.round(df['humidity'].std(), 2)
humidity_mean, humidity_std
```

(61.89, 19.25)

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

```
# The below code generates a histogram plot for the 'humidity' feature, showing the cumulative
    # distribution of humidity values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'humidity', kde = True, cumulative = True, stat = 'percent')
plt.grid(axis = 'y', linestyle = '--')    # setting the gridlines along y axis
plt.yticks(np.arange(0, 101, 10))
plt.plot()       # displaying the chart
```

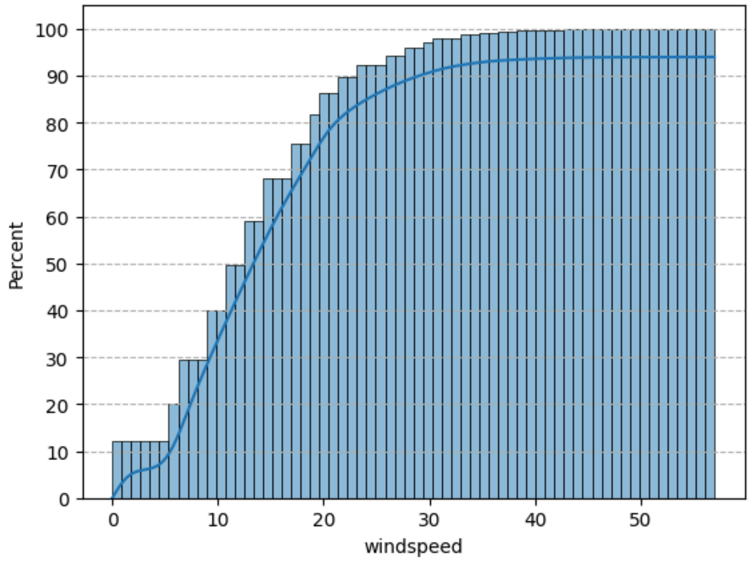
[]



- More than 80 % of the time, the humidity value is greater than 40. Thus for most of the time, humidity level varies from optimum to too moist.

```
sns.histplot(data = df, x = 'windspeed', kde = True, cumulative = True, stat = 'percent')
plt.grid(axis = 'y', linestyle = '--')
plt.yticks(np.arange(0, 101, 10))
plt.plot()       # displaying the chart
```

[]



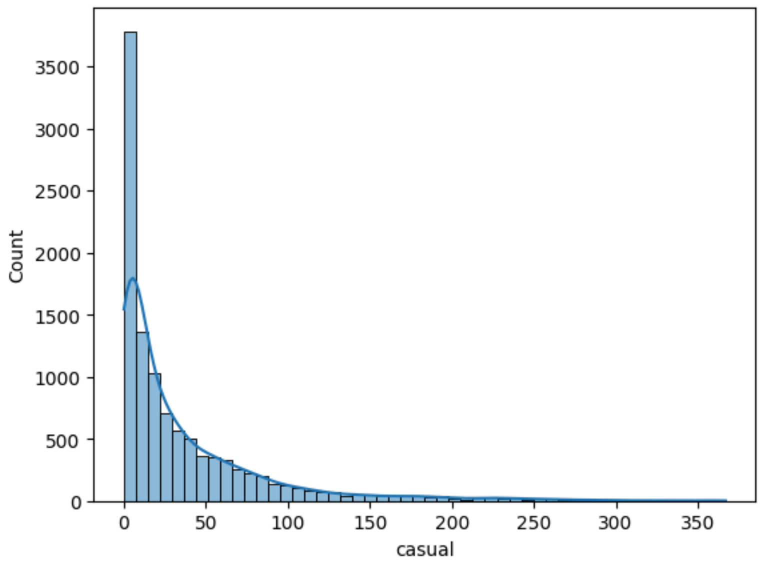
- More than 85 % of the total windspeed data has a value of less than 20.

```
len(df[df['windspeed'] < 20]) / len(df)
```

```
0.8626676465184641
```

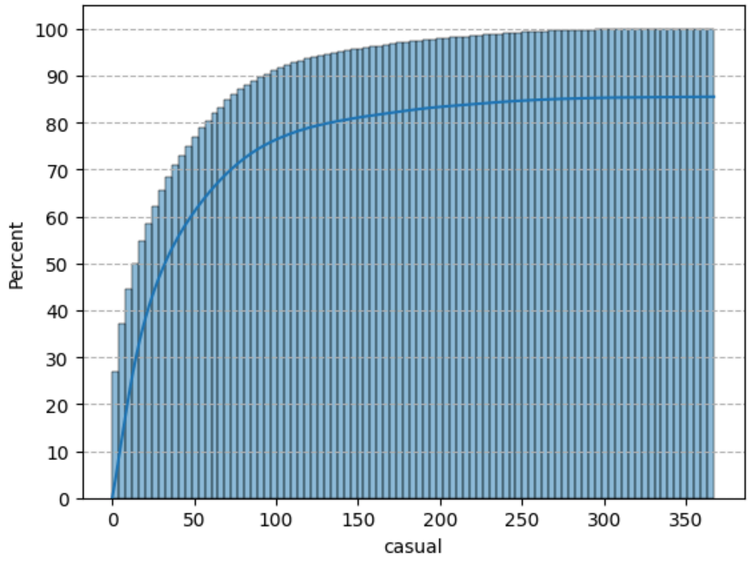
```
# The below code generates a histogram plot for the 'casual' feature, showing the distribution of
    # casual users' values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'casual', kde = True, bins = 50)
plt.plot()      # displaying the chart
```

[]




```
sns.histplot(data = df, x = 'casual', kde = True, cumulative = True, stat = 'percent')
plt.grid(axis = 'y', linestyle = '--')
plt.yticks(np.arange(0, 101, 10))
plt.plot()        # displaying the chart
```

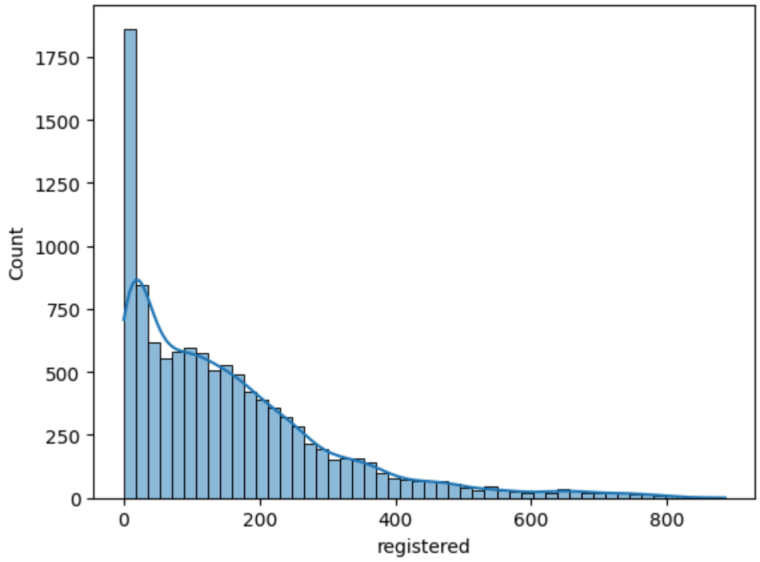
[]



- More than 80 % of the time, the count of casual users is less than 60.

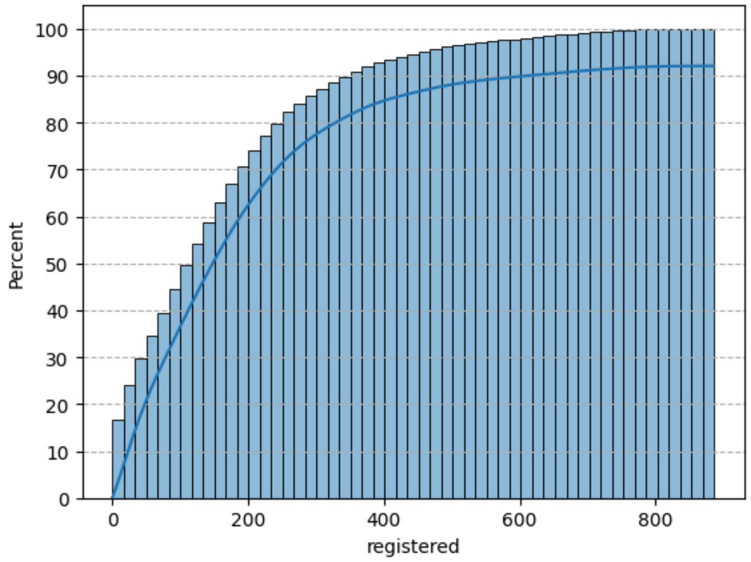
```
# The below code generates a histogram plot for the 'registered' feature, showing the distribution of
    # registered users' values in the dataset.
# The addition of the kernel density estimation plot provides
    # a visual representation of the underlying distribution shape, making it easier to analyze the
    # data distribution.

sns.histplot(data = df, x = 'registered', kde = True, bins = 50)
plt.plot()          # displaying the chart
```

[]




```
sns.histplot(data = df, x = 'registered', kde = True, cumulative = True, stat = 'percent')
plt.grid(axis = 'y', linestyle = '--')
plt.yticks(np.arange(0, 101, 10))
plt.plot()        # displaying the chart
```

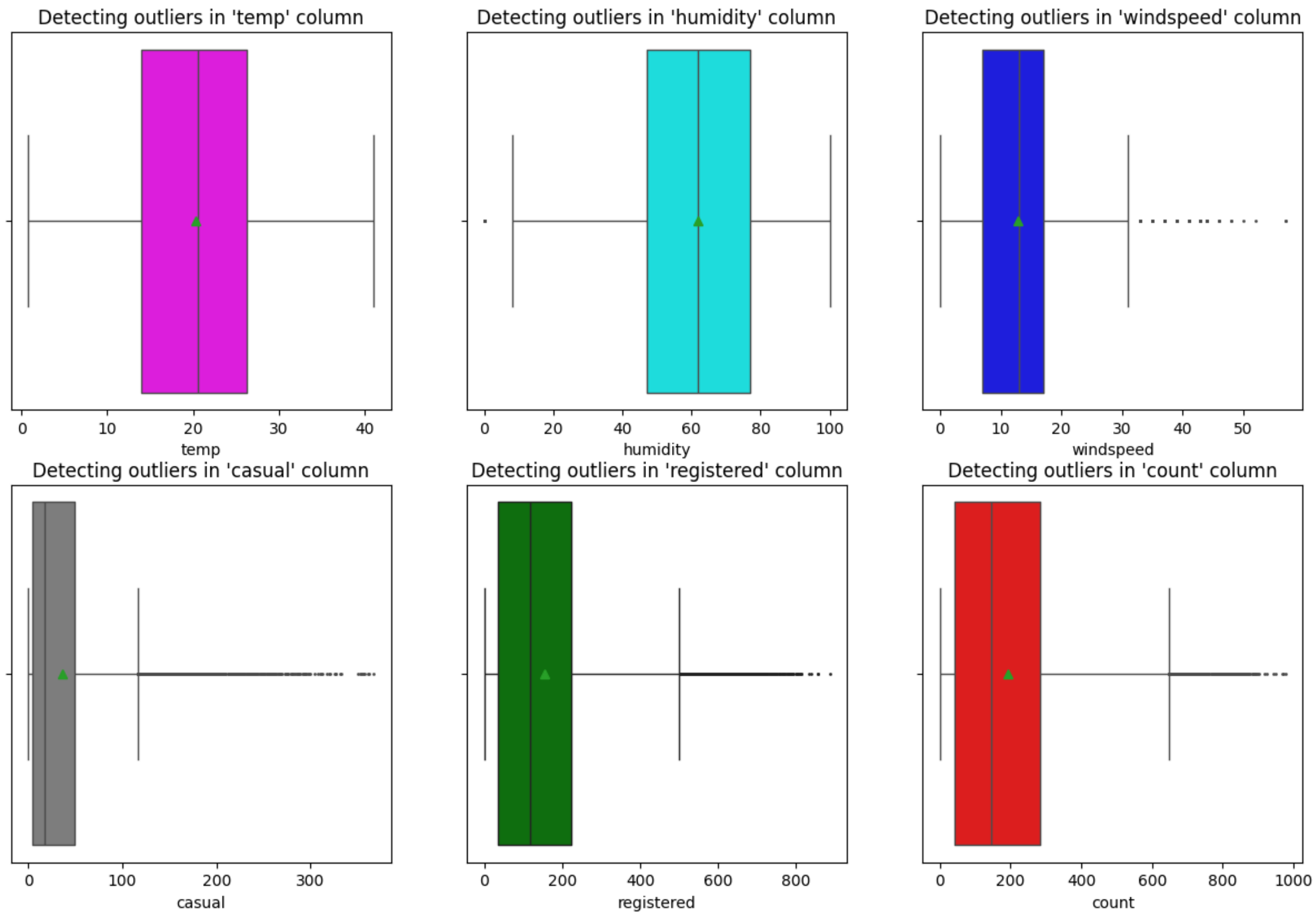
[]



- More than 85 % of the time, the count of registered users is less than 300.

Outliers Detection

```
columns = ['temp', 'humidity', 'windspeed', 'casual', 'registered', 'count']
colors = np.random.permutation(['red', 'blue', 'green', 'magenta', 'cyan', 'gray'])
count = 1
plt.figure(figsize = (15, 15))
for i in columns:
    plt.subplot(3, 3, count)
    plt.title(f"Detecting outliers in '{i}' column")
    sns.boxplot(data = df, x = df[i], color = colors[count - 1], showmeans = True, fliersize = 1)
    plt.plot()
    count += 1
```

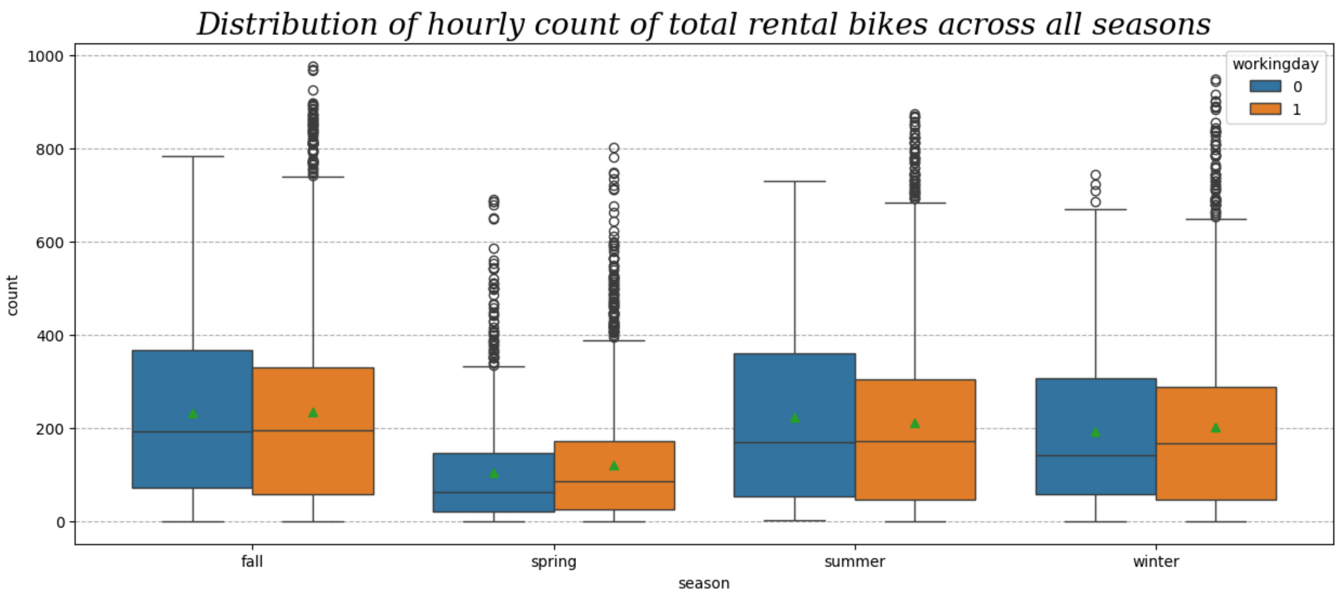


- There is no outlier in the temp column.
- There are few outliers present in humidity column.
- There are many outliers present in each of the columns : windspeed, casual, registered, count.

Bivariate Analysis

```
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.plot()
```

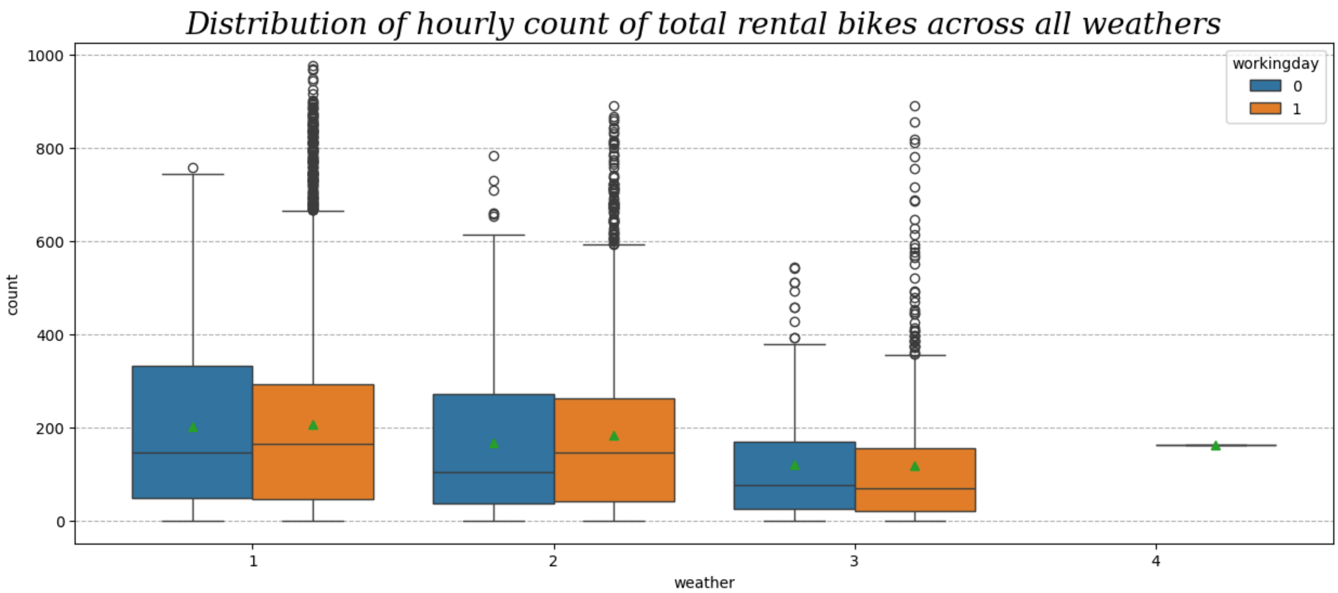
[]



- The hourly count of total rental bikes is higher in the fall season, followed by the summer and winter seasons. It is generally low in the spring season.

```
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.plot()
```

[]



- The hourly count of total rental bikes is higher in the clear and cloudy weather, followed by the misty weather and rainy weather. There are very few records for extreme weather conditions.

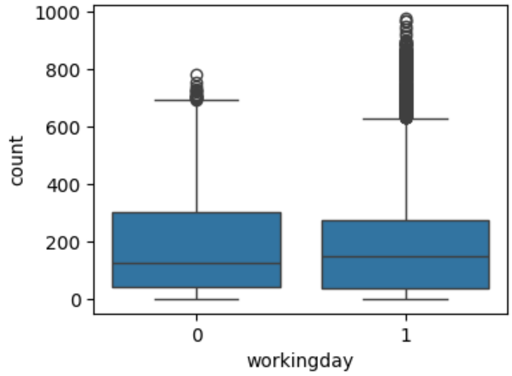
1)Is there any effect of Working Day on the number of electric cycles rented ?

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

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

```
plt.figure(figsize = (4, 3))
sns.boxplot(data = df, x = 'workingday', y = 'count')
plt.plot()
```

[]



STEP-1 : Set up Null Hypothesis

- **Null Hypothesis (H0)** - Working Day does not have any effect on the number of electric cycles rented.
- **Alternate Hypothesis (HA)** - Working Day has some effect on the number of electric cycles rented

STEP-2 : Checking for basic assumpitons 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 T Test are met then we can proceed performing T Test for independent samples else we will perform the non parametric test equivalent to T Test for independent sample i.e., Mann-Whitney U rank test for two independent samples.

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

- We set our ***alpha to be 0.05***

STEP-5: Compare p-value and alpha.

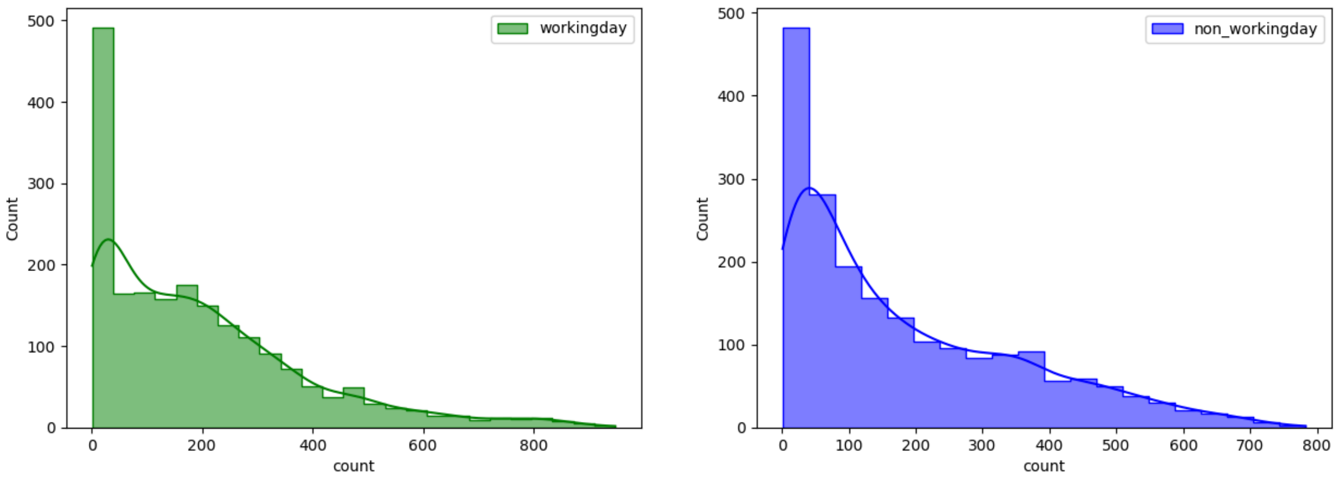
- Based on p-value, we will accept or reject H0.

1. **p-val > alpha** : Accept H0
2. **p-val < alpha** : Reject H0

Visual Tests to know if the samples follow normal distribution

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
sns.histplot(df.loc[df['workingday'] == 1, 'count'].sample(2000),
             element = 'step', color = 'green', kde = True, label = 'workingday')
plt.legend()
plt.subplot(1, 2, 2)
sns.histplot(df.loc[df['workingday'] == 0, 'count'].sample(2000),
             element = 'step', color = 'blue', kde = True, label = 'non_workingday')
plt.legend()
plt.plot()
```

[]

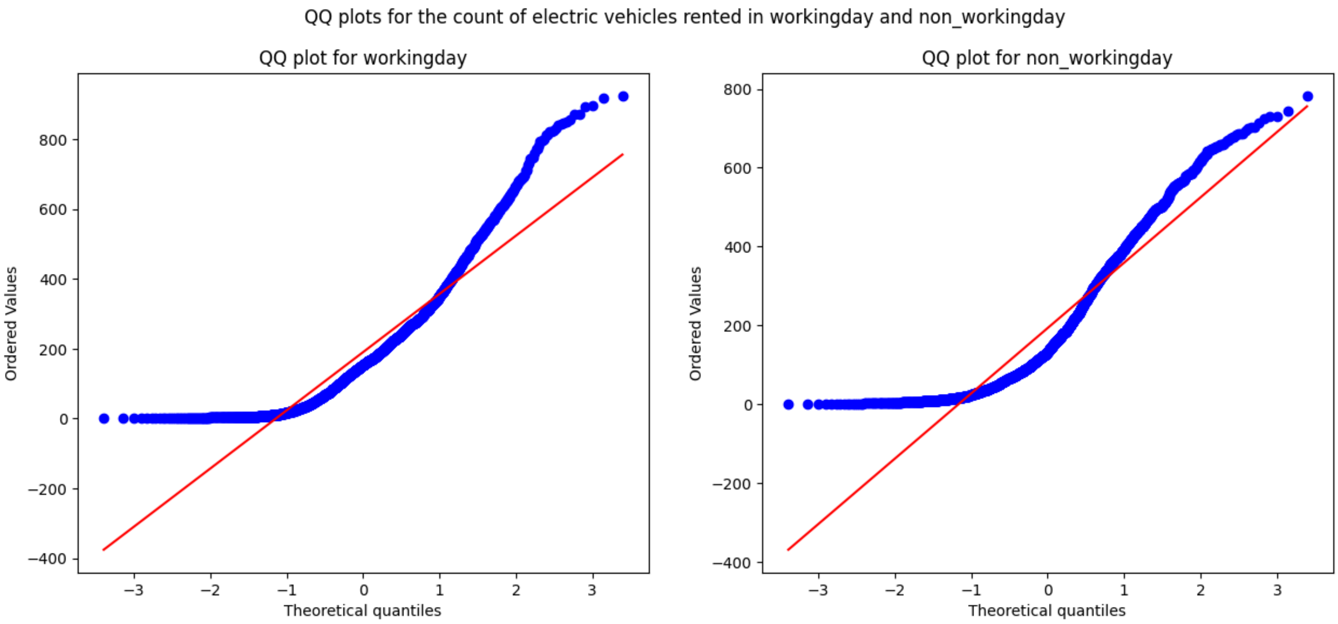


- It can be inferred from the above plot that the distributions do not follow normal distribution.

Distribution check using QQ Plot

```
plt.figure(figsize = (15, 6))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for the count of electric vehicles rented in workingday and non_workingday')
spy.probplot(df.loc[df['workingday'] == 1, 'count'].sample(2000), plot = plt, dist = 'norm')
plt.title('QQ plot for workingday')
plt.subplot(1, 2, 2)
spy.probplot(df.loc[df['workingday'] == 0, 'count'].sample(2000), plot = plt, dist = 'norm')
plt.title('QQ plot for non_workingday')
plt.plot()
```

[]



- It can be inferred from the above plot that the distributions do not follow normal distribution.

It can be seen from the above plots that the samples do not come from normal distribution.

- Applying Shapiro-Wilk test for normality

H_0 : The sample **follows normal distribution** H_1 : The sample **does not follow normal distribution**

alpha = 0.05

Test Statistics : **Shapiro-Wilk test for normality**

```
test_stat, p_value = spy.shapiro(df.loc[df['workingday'] == 1, 'count'].sample(2000))
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 2.3672631017495506e-38
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df.loc[df['workingday'] == 0, 'count'].sample(2000))
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 3.913803562302073e-36
The sample does not follow normal distribution
```

Transforming the data using boxcox transformation and checking if the transformed data follows normal distribution.

```
import warnings

# Assuming you have loaded your data into a DataFrame 'df'
transformed_workingday = spy.boxcox(df.loc[df['workingday'] == 1, 'count'])[0]

# Ignore the UserWarning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    test_stat, p_value = spy.shapiro(transformed_workingday)

print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows a normal distribution')
```

```
p-value 1.6136246052607705e-33
The sample does not follow normal distribution
```

```
transformed_non_workingday = spy.boxcox(df.loc[df['workingday'] == 1, 'count'])[0]

# Ignore the UserWarning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    test_stat, p_value = spy.shapiro(transformed_workingday)

print('p-value', p_value)
if p_value < 0.05:
    print('The sample does not follow normal distribution')
else:
    print('The sample follows a normal distribution')
```

```
p-value 1.6136246052607705e-33
The sample does not follow normal distribution
```

- Even after applying the boxcox transformation on each of the "workingday" and "non_workingday" data, the samples do not follow normal distribution.

- Homogeneity of Variances using **Lavene's test**

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(df.loc[df['workingday'] == 1, 'count'].sample(2000),
                                df.loc[df['workingday'] == 0, 'count'].sample(2000))
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 0.8680486700037203
The samples have Homogenous Variance
```

Since the samples are not normally distributed, T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

```
# Ho : Mean no.of electric cycles rented is same for working and non-working days
# Ha : Mean no.of electric cycles rented is not same for working and non-working days
# Assuming significance Level to be 0.05
# Test statistics : Mann-Whitney U rank test for two independent samples

test_stat, p_value = spy.mannwhitneyu(df.loc[df['workingday'] == 1, 'count'],
                                      df.loc[df['workingday'] == 0, 'count'])
print('P-value :',p_value)
if p_value < 0.05:
    print('Mean no.of electric cycles rented is not same for working and non-working days')
else:
    print('Mean no.of electric cycles rented is same for working and non-working days')
```

```
P-value : 0.9679139953914079
Mean no.of electric cycles rented is same for working and non-working days
```

Therefore, the mean hourly count of the total rental bikes is statistically same for both working and non- working days .

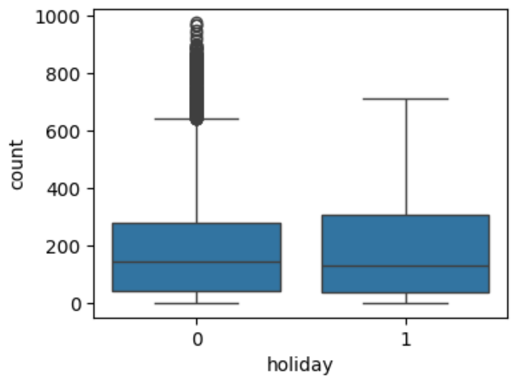
2)Is there any effect of holidays on the number of electric cycles rented ?

```
df.groupby(by = 'holiday')['count'].describe()
```

holiday	count	mean	std	min	25%	50%	75%	max
0	10575.0	191.741655	181.513131	1.0	43.0	145.0	283.0	977.0
1	311.0	185.877814	168.300531	1.0	38.5	133.0	308.0	712.0

```
plt.figure(figsize = (4, 3))
sns.boxplot(data = df, x = 'holiday', y = 'count')
plt.plot()
```

```
[]
```



STEP-1 : Set up Null Hypothesis

- **Null Hypothesis (H0)** - Holidays have no effect on the number of electric vehicles rented
- **Alternate Hypothesis (HA)** - Holidays has some effect on the number of electric vehicles rented

STEP-2 : Checking for basic assumpitons 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 T Test are met then we can proceed performing T Test for independent samples else we will perform the non parametric test equivalent to T Test for independent sample i.e., Mann-Whitney U rank test for two independent samples.

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

- We set our ***alpha to be 0.05***

STEP-5: Compare p-value and alpha.

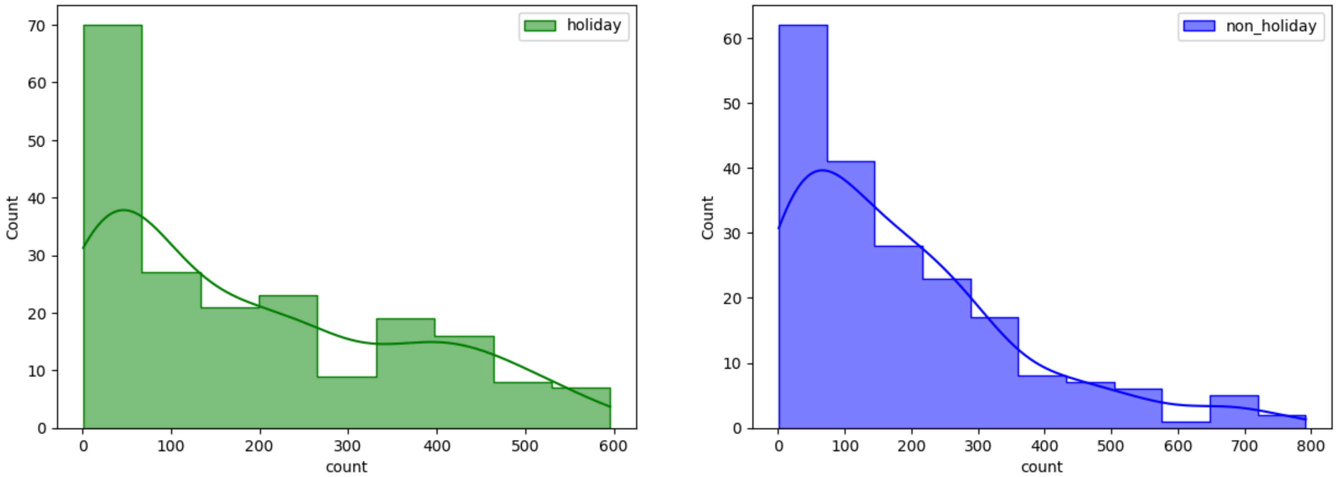
- Based on p-value, we will accept or reject H0.

1. **p-val > alpha** : Accept H0
2. **p-val < alpha** : Reject H0

Visual Tests to know if the samples follow normal distribution

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
sns.histplot(df.loc[df['holiday'] == 1, 'count'].sample(200),
             element = 'step', color = 'green', kde = True, label = 'holiday')
plt.legend()
plt.subplot(1, 2, 2)
sns.histplot(df.loc[df['holiday'] == 0, 'count'].sample(200),
             element = 'step', color = 'blue', kde = True, label = 'non_holiday')
plt.legend()
plt.plot()
```

[]

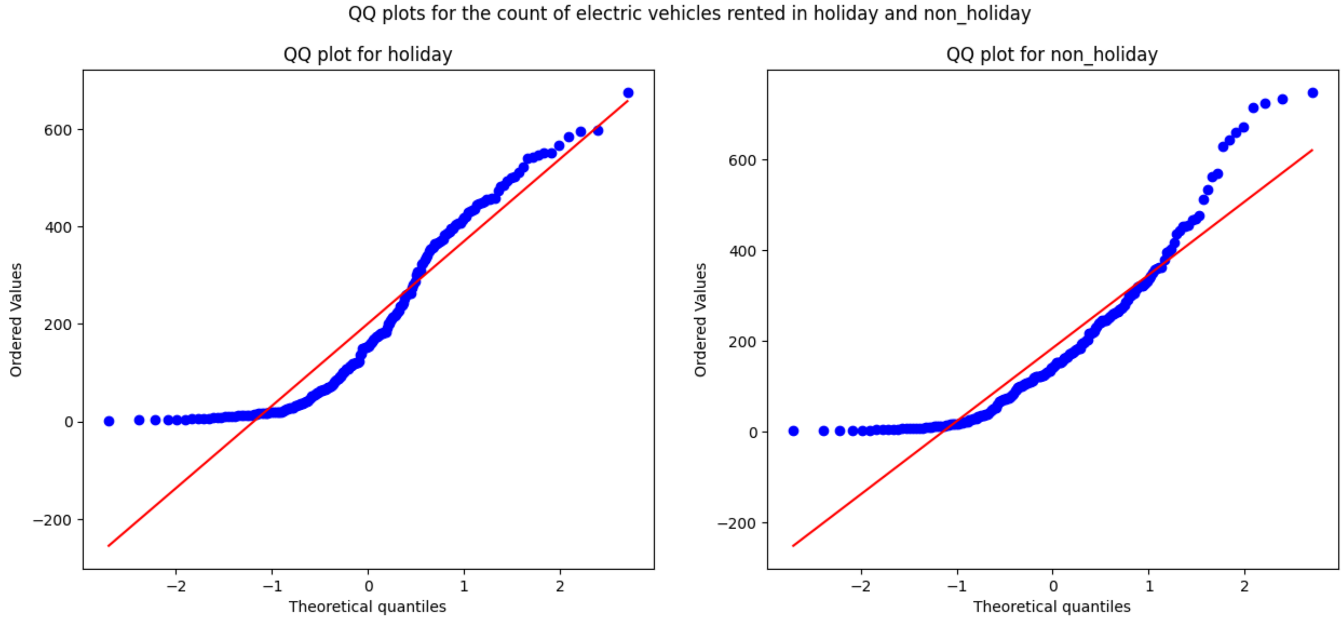


- It can be inferred from the above plot that the distributions do not follow normal distribution.

Distribution check using QQ Plot

```
plt.figure(figsize = (15, 6))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for the count of electric vehicles rented in holiday and non_holiday')
spy.probplot(df.loc[df['holiday'] == 1, 'count'].sample(200), plot = plt, dist = 'norm')
plt.title('QQ plot for holiday')
plt.subplot(1, 2, 2)
spy.probplot(df.loc[df['holiday'] == 0, 'count'].sample(200), plot = plt, dist = 'norm')
plt.title('QQ plot for non_holiday')
plt.plot()
```

[]



- It can be inferred from the above plot that the distributions do not follow normal distribution.

It can be seen from the above plots that the samples do not come from normal distribution.

- Applying Shapiro-Wilk test for normality

H_0 : The sample **follows normal distribution** H_1 : The sample **does not follow normal distribution**

alpha = 0.05

Test Statistics : **Shapiro-Wilk test for normality**

```
test_stat, p_value = spy.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')
```

```
p-value 1.4502229972457314e-10
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.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 1.4108062591430826e-12
The sample does not follow normal distribution
```

Transforming the data using boxcox transformation and checking if the transformed data follows normal distribution.

```
transformed_holiday = spy.boxcox(df.loc[df['holiday'] == 1, 'count'])[0]
test_stat, p_value = spy.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')
```

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

```
transformed_non_holiday = spy.boxcox(df.loc[df['holiday'] == 0, 'count'].sample(5000))[0]
test_stat, p_value = spy.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 1.2853873385589348e-25
The sample does not follow normal distribution
```

- Even after applying the boxcox transformation on each of the "holiday" and "non_holiday" data, the samples do not follow normal distribution.

Homogeneity of Variances using Levene's test

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.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 0.4194866005904715
The samples have Homogenous Variance
```

Since the samples are not normally distributed, T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

```
# 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 = spy.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 days')
else:
    print('No.of electric cycles rented is similar for holidays and non-holidays')
```

```
P-value : 0.33504616396098297
No.of electric cycles rented is similar for holidays and non-holidays
```

Therefore, the number of electric cycles rented is statistically similar for both holidays and non - holidays.

3)Is weather dependent on the season ?

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

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

- It is clear from the above statistical description that both 'weather' and 'season' features are categorical in nature.

STEP-1 : Set up Null Hypothesis

1. **Null Hypothesis (H0)** - weather is independent of season
2. **Alternate Hypothesis (HA)** - weather is dependent of 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 **Chi-Square Distribution**.

STEP-3: Checking for basic assumptons 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 **value of the cell expecteds should be 5 or more** in at least 80% of the cells, and no cell should have an expected of less than one (3).

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

---

we will be computing the chi square-test p-value using the chi2_contingency function using scipy.stats. We set our **alpha to be 0.05**

STEP-5: Compare p-value and alpha.

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

1. **p-val > alpha** : Accept H0
2. **p-val < 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.

```
# First, finding the contingency table such that each value is the total number of total bikes rented
  # 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
```

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

Next steps: Generate code with `cross_table` View recommended plots

Since the above contingency table has one column in which the count of the rented electric vehicle is less than 5 in most of the cells, we can remove the weather 4 and then proceed further.

```
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
```

```
array([[470116, 139386,  31160],
       [223009,  76406,  12919],
       [426350, 134177,  27755],
       [356588, 157191,  30255]])
```

```
chi_test_stat, p_value, dof, expected = spy.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

```
#Assume signifance level = 5%
alpha = 0.05
if p_value < alpha:
    print('Reject Null Hypothesis')
else:
    print('Failed to reject Null Hypothesis')
```

```
Failed to reject Null Hypothesis
```

Therefore, there is statistically significant dependency of weather and season based on the number of number of bikes rented.

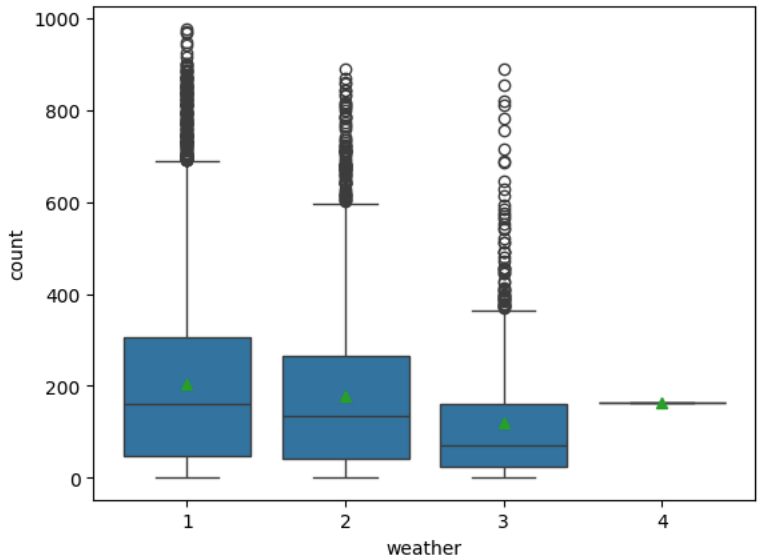
4)Is the number of cycles rented is similar or different in different weather ?

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

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

```
sns.boxplot(data = df, x = 'weather', y = 'count', showmeans = True)
plt.plot()
```

```
[]
```




```
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)
```

```
(7192, 2834, 859, 1)
```

STEP-1 : Set up Null Hypothesis

- **Null Hypothesis (H0)** - Mean of cycle rented per hour is same for weather 1, 2 and 3. (We wont be considering weather 4 as there in only 1 data point for weather 4 and we cannot perform a ANOVA test with a single data point for a group)
- **Alternate Hypothesis (HA)** -Mean of cycle rented per hour is not same for season 1,2,3 and 4 are different.

STEP-2 : Checking for basic assumpitons for the hypothesis

Normality check using **QQ Plot**. If the distribution is not normal, use **BOX-COX transform** to transform it to normal distribution.

Homogeneity of Variances using **Levene's test**

Each observations are **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=MSB / MSW

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

STEP-4: Decide the kind of test.

We will be performing **right tailed f-test**

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

---

we will be computing the anova-test p-value using the f_oneway function using scipy.stats. We set our **alpha to be 0.05**

STEP-6: Compare p-value and alpha.

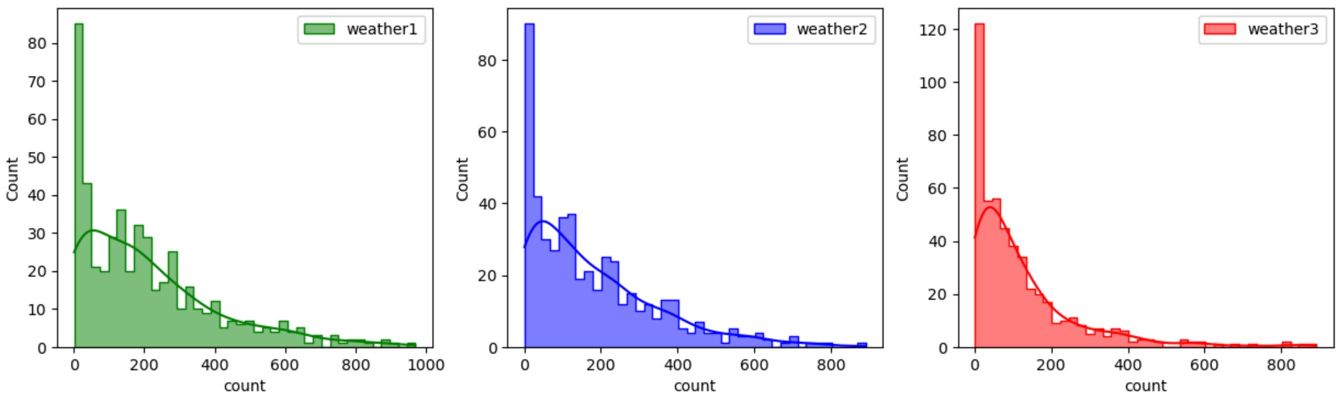
Based on p-value, we will accept or reject H0.

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

Visual Tests to know if the samples follow normal distribution

```
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()
```

[]

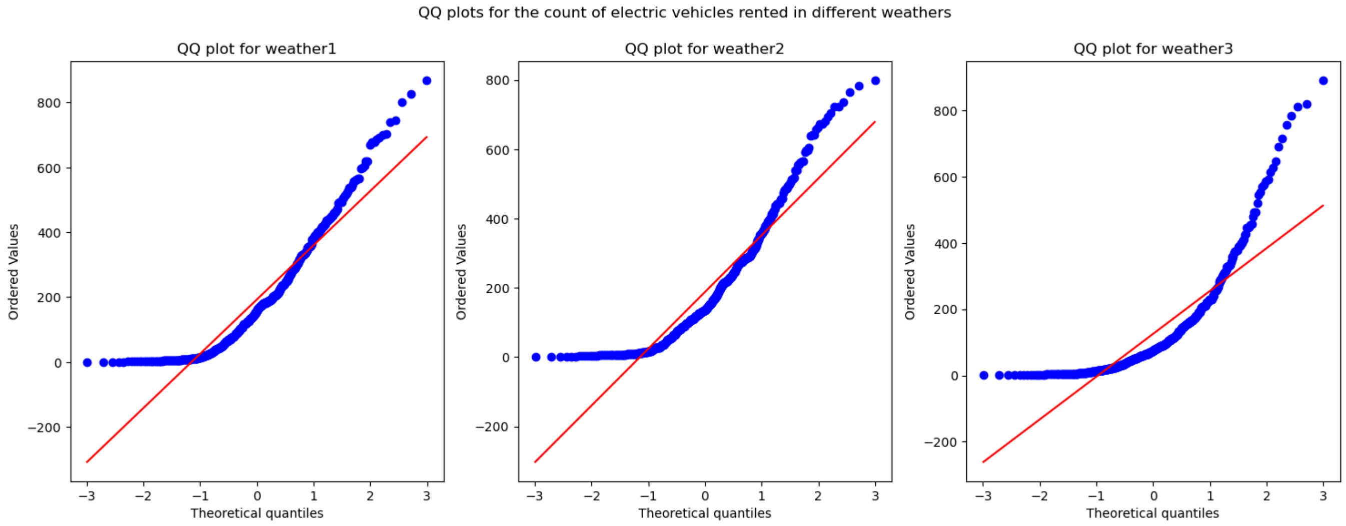


- It can be inferred from the above plot that the distributions do not follow normal distribution.

Distribution check using QQ Plot

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

```
[]
```



- It can be inferred from the above plot that the distributions do not follow normal distribution.

It can be seen from the above plots that the samples do not come from normal distribution.

- Applying Shapiro-Wilk test for normality

H_0 : The sample **follows normal distribution** H_1 : The sample **does not follow normal distribution**

alpha = 0.05

Test Statistics : **Shapiro-Wilk test for normality**

```
test_stat, p_value = spy.shapiro(df_weather1.loc[:, 'count'].sample(500))
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 3.044791219012407e-16
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df_weather2.loc[:, 'count'].sample(500))
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 2.673580605657277e-19
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df_weather3.loc[:, 'count'].sample(500))
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 1.2996631324935898e-25
The sample does not follow normal distribution
```

Transforming the data using boxcox transformation and checking if the transformed data follows normal distribution.

```
transformed_weather1 = spy.boxcox(df_weather1.loc[:, 'count'].sample(5000))[0]
test_stat, p_value = spy.shapiro(transformed_weather1)
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 3.599291124751596e-27
The sample does not follow normal distribution
```

```
transformed_weather2 = spy.boxcox(df_weather2.loc[:, 'count'])[0]
test_stat, p_value = spy.shapiro(transformed_weather2)
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 1.9219748327822736e-19
The sample does not follow normal distribution
```

```
transformed_weather3 = spy.boxcox(df_weather3.loc[:, 'count'])[0]
test_stat, p_value = spy.shapiro(transformed_weather3)
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 1.4137293646854232e-06
The sample does not follow normal distribution
```

- Even after applying the boxcox transformation on each of the weather data, the samples do not follow normal distribution.

Homogeneity of Variances using Levene's test

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.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  Homogenous Variance')
else:
    print('The samples have Homogenous Variance ')
```

```
p-value 5.963299306183821e-11
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.

```
# 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 = spy.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

```
#Assume signifance level = 5%
alpha = 0.05
if p_value.all() < 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 weathers.

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

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

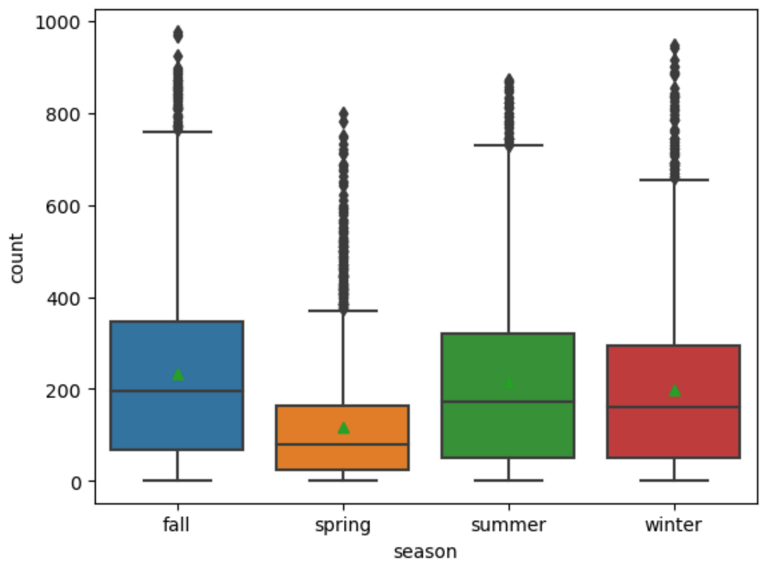
season	count	mean	std	min	25%	50%	75%	max
fall	2733.0	234.417124	197.151001	1.0	68.0	195.0	347.0	977.0
spring	2686.0	116.343261	125.273974	1.0	24.0	78.0	164.0	801.0
summer	2733.0	215.251372	192.007843	1.0	49.0	172.0	321.0	873.0
winter	2734.0	198.988296	177.622409	1.0	51.0	161.0	294.0	948.0

```
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_winter)
```

```
(2686, 2733, 2733, 2734)
```

```
sns.boxplot(data = df, x = 'season', y = 'count', showmeans = True)
plt.plot()
```

```
[]
```



STEP-1 : Set up Null Hypothesis

- **Null Hypothesis (H0)** - Mean of cycle rented per hour is same for season 1,2,3 and 4.

- **Alternate Hypothesis (HA)** -Mean of cycle rented per hour is different for season 1,2,3 and 4.

STEP-2 : Checking for basic assumpitons for the hypothesis

---

1. **Normality check** using QQ Plot. If the distribution is not normal, use **BOX-COX transform** to transform it to normal distribution.
2. Homogeneity of Variances using **Levene's test**
3. Each observations are **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=MSB/MSW

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

STEP-4: Decide the kind of test.

We will be performing **right tailed f-test**

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

---

we will be computing the anova-test p-value using the **f_oneway** function using scipy.stats. We set our alpha to be **0.05**

STEP-6: Compare p-value and alpha.

Based on p-value, we will accept or reject H0. p-val > alpha : Accept H0 p-val < 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):

μ1 = μ2 = μ3 = ... = μk

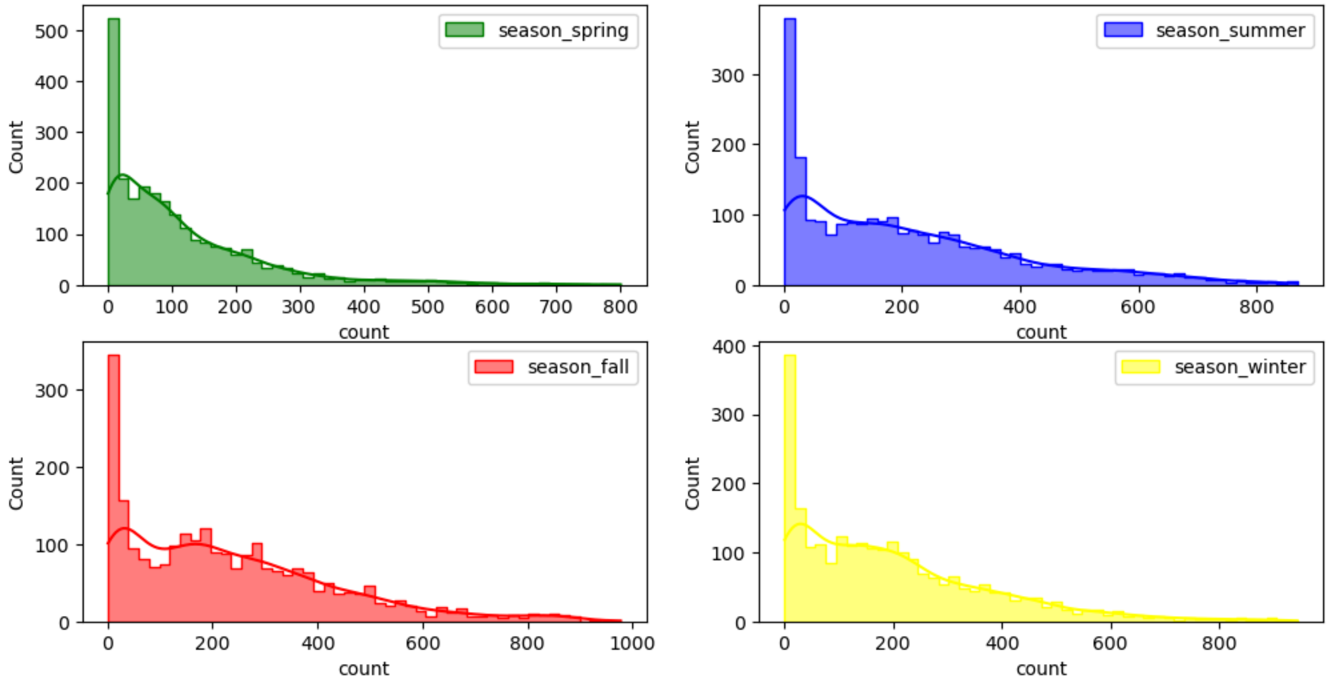
where, μ = group mean and k = number of groups.

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

```
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()
```

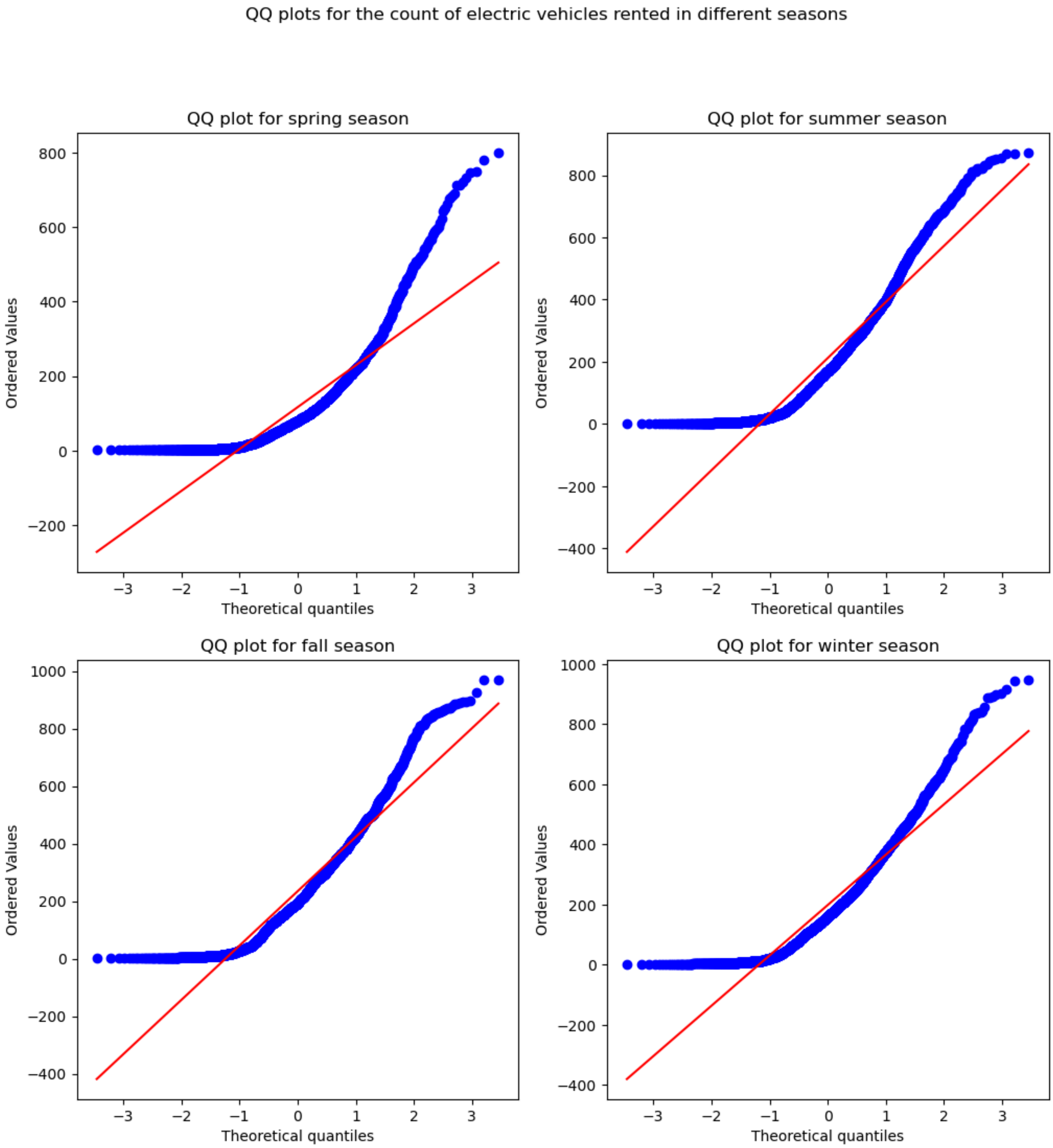
[ ]



- It can be inferred from the above plot that the distributions do not follow normal distribution.

Distribution check using QQ Plot

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

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

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

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



- It can be inferred from the above plots that the distributions do not follow normal distribution.

It can be seen from the above plots that the samples do not come from normal distribution.

- Applying Shapiro-Wilk test for normality

H_0 : The sample **follows normal distribution** H_1 : The sample **does not follow normal distribution**

alpha = 0.05

Test Statistics : **Shapiro-Wilk test for normality**

```
test_stat, p_value = spy.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')
```

```
p-value 0.0
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.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')
```

```
p-value 2.754738886664996e-37
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.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')
```

```
p-value 2.7709185612358606e-35
The sample does not follow normal distribution
```

```
test_stat, p_value = spy.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')
```

```
p-value 3.220378090985585e-38
The sample does not follow normal distribution
```

Transforming the data using boxcox transformation and checking if the transformed data follows normal distribution.

```
transformed_df_season_spring = spy.boxcox(df_season_spring.sample(2500))[0]
test_stat, p_value = spy.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')
```

```
p-value 2.4352035106161074e-17
The sample does not follow normal distribution
```

```
transformed_df_season_summer = spy.boxcox(df_season_summer.sample(2500))[0]
test_stat, p_value = spy.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')
```

```
p-value 5.329487124278294e-21
The sample does not follow normal distribution
```

```
transformed_df_season_fall = spy.boxcox(df_season_fall.sample(2500))[0]
test_stat, p_value = spy.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')
```

```
p-value 1.6579020872325687e-21
The sample does not follow normal distribution
```

```
transformed_df_season_winter = spy.boxcox(df_season_winter.sample(2500))[0]
test_stat, p_value = spy.shapiro(transformed_df_season_winter)
print('p-value', p_value)
if p_value < 0.05:
```